ใบรับรองวิทยานิพนธ์

บัณฑิตวิทยาลัย มหาวิทยาลัยเกษตรศาสตร์

วิศวกรรมศาสตรมหาบัณฑิต (วิศวกรรมคอมพิวเตอร์)

ปริญญา

วิศวกรรมคอมพิวเตอร์ วิศวกรรมคอมพิวเตอร์

สาขา **ภาควิชา**

เรื่อง เฟรมเวิร์กและเทคนิคการคอมไพล์สำหรับการพัฒนาโปรแกรมบนเครื่องจีพียู คลัสเตอร์ด้วยการใช้ตัวชี้แนะคอมไพเลอร์

The Framework and Compilation Techniques for Directive-based GPU Cluster Programmi

นามผู้วิจัย นายพิสิษฐ์ มรรคไพสิฐ

ได้พิจารณาเห็นชอบโดย

อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก .....

(ผู้ช่วยศาสตราจารย์ภุชงค์ อุทโยภาศ, Ph.D.)

อาจารย์ที่ปรึกษาวิทยานิพนธ์ร่วม .....

(อาจารย์ภารุจ รัตนวรพันธุ์, D.Eng.)

หัวหน้าภาควิชา .....

(รองศาสตราจารย์อนันต์ ผลเพิ่ม, Ph.D.)

บัณฑิตวิทยาลัย มหาวิทยาลัยเกษตรศาสตร์รับรองแล้ว

.....

(รองศาสตราจารย์กัญจนา ธีระกุล, D.Agr.)

คณบดีบัณฑิตวิทยาลัย

วันที่ ..... **เดือน** ..... **พ.ศ.** .....

วิทยานิพนธ์

เรื่อง

เฟรมเวิร์กและเทคนิคการคอมไพล์สำหรับการพัฒนาโปรแกรมบนเครื่องจีพียูคลัสเตอร์ด้วยการใช้ตัวชี้แนะคอมไพเลอร์

The Framework and Compilation Techniques for Directive-based GPU Cluster Programming

โดย

นายพิสิษฐ์ มรรคไพสิฐ

เสนอ

บัณฑิตวิทยาลัย มหาวิทยาลัยเกษตรศาสตร์
เพื่อความสมบูรณ์แห่งปริญญาวิศวกรรมศาสตรมหาบัณฑิต (วิศวกรรมคอมพิวเตอร์)
พ.ศ. 2557

พิสิษฐ์ มรรคไพสิฐ 2557: เฟรมเวิร์กและเทคนิคการคอมไพล์สำหรับการพัฒนาโปรแกรมบนเครื่องจีพียูคลัสเตอร์ด้วยการใช้ตัวชี้แนะคอมไพเลอร์ ปริญญาวิศวกรรมศาสตรมหาบัณฑิต (วิศวกรรมคอมพิวเตอร์) สาขาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์ อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก: ผู้ช่วยศาสตราจารย์ภุชงค์ อุทโยภาศ, Ph.D. 62 หน้า

จีพียูคลัสเตอร์เป็นสถาปัตยกรรมของเครื่องคอมพิวเตอร์ที่นิยมใช้ในการคำนวนเชิงวิทยาศาสตร์และวิศวกรรมศาสตร์ขนาดใหญ่ อย่างไรก็ตามการพัฒนาโปรแกรมบนสถาปัตยกรรมนี้ยังคงเป็นงานที่ยากและซับซ้อน เพื่อแก้ปัญหานี้ผู้วิจัยได้นำเอาตัวชี้แนะคอมไพเลอร์มาตรฐานโอเพนเอซีซีมาปรับปรุงเพิ่มเติมเพื่อให้สามารถปรับใช้กับสถาปัตยกรรมแบบจีพียูคลัสเตอร์ได้ งานวิจัยนี้ได้เสนอส่วนต่อขยายของโอเพนเอซีซีที่จะช่วยให้สามารถกำหนดรูปแบบการกระจายข้อมูล และความขึ้นต่อกันของข้อมูลในแต่ละโหนดของเครื่องคลัสเตอร์ ขั้นตอนการแปลงโค้ดจากตัวชี้แนะคอมไพเลอร์ที่นำเสนอไปยังโค้ดที่สามารถทำงานบนจีพียูคลัสเตอร์ได้ สถาปัตยกรรมสำหรับการคอมไพล์และรันโปรแกรม รวมไปถึงเทคนิคการปรับปรุงสมรรถนะด้วยการแบ่งการทำงานบนจีพียูเป็นหลายส่วน เพื่อให้เวลาการทำงานพร้อมกันระหว่างการส่งข้อมูลและการประมวลผลมีค่าสูงสุด ซึ่งจะทำให้เวลาการประมวลผลลดลงได้เป็นอย่างมาก ผลการทดลองได้แสดงให้เห็นว่าตัวชี้แนะคอมไพเลอร์และชุดเครื่องมือที่นำเสนอนั้นสามารถลดเวลาการพัฒนาโปรแกรมขนาดใหญ่บนเครื่องจีพียูคลัสเตอร์ไปได้มาก โดยที่ยังคงประสิทธิภาพของการประมวลผลเอาไว้

_____ _____ ___ / ___ / ___

ลายมือชื่อนิสิต ลายมือชื่ออาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก

Pisit Makpaisit 2014: The Framework and Compilation Techniques for Directive-based GPU Cluster Programming. Master of Engineering (Computer Engineering), Major Field: Computer Engineering, Department of Computer Engineering.
Thesis Advisor: Assistant Professor Putchong Uthayophas, Ph.D. 62 pages.

GPU cluster is an important architecture being used for large scientific and engineering applications. However, manually developed GPU cluster application is still a very difficult task. To alleviate this problem, we adopt the OpenACC standard for directive-based approach and proposed some extension to support GPU cluster programming. The extensions are clauses used to define the memory distribution and dependency of tasks on cluster nodes. We propose framework and technique used to implement a source-to-source compiler to support the proposed extension. We also propose a kernel split technique for automatic optimization in our compiler, The experiment conducted on the source code translation tool developed in this work show that the speedup close to hand code can be achieved on commonly used scientific application with much less programming effort.

_____ _____ ___ / ___ / ___

Student's signature Thesis Advisor's signature

กิตติกรรมประกาศ

งานวิจัยชิ้นนี้เป็นผลจากประสบการณ์ของข้าพเจ้าและการทำงานอย่างยาวนาน ดังนั้นจึงยากที่จะไม่มีผู้มีส่วนเกี่ยวข้องในงานชิ้นนี้ ข้าพเจ้าจึงอยากใช้พื้นที่ส่วนนี้เพื่อเป็นการกล่าวคำขอบคุณบุคคลที่ได้ช่วยเหลือให้งานวิจัยนี้สำเร็จลุล่วงไปได้

ขอขอบคุณผู้ช่วยศาสตราจารย์ ดร.ภุชงค์ อุทโยภาศ อาจารย์ที่ปรึกษางานวิจัย ที่ได้มอบแนวทางการวิจัยและโอกาสการทำงานมากมายแก่ข้าพเจ้า การได้พูดคุยและรับฟังคำพูดของอาจารย์ทำให้ข้าพเจ้าได้รับรู้เรื่องราวที่กว้างขวาง เข้าใจเรื่องราวในโลกมากขึ้น ให้ความสำคัญกับคุณธรรมเป็นอันดับแรก และยังเปิดทัศนคติใหม่มากมายแก่ข้าพเจ้า ขอบคุณรองศาสตราจารย์ ดร.จันทนา จันทราพรชัย และอาจารย์ภารุจ รัตนบรพันธุ์ ในคำปรึกษาการทำงานวิจัยชิ้นนี้ของข้าพเจ้า ขอบคุณอาจารย์ทุกท่านที่ช่วย ประสิทธิ์ประสาท วิชาจนทำให้ข้าพเจ้าสามารถทำงานวิจัยนี้ได้ โดยเฉพาะอย่างยิ่ง ดร.วรวรรณ ดีอัซ การ์บาโย ผู้ซึ่งให้คำแนะนำและคำปรึกษาแก่ข้าพเจ้าในทุกๆเรื่อง

ขอบคุณรุ่นพี่และรุ่นน้องสมาชิกทุกคนในห้องปฏิบัติการ ศูนย์วิจัยระบบคอมพิวเตอร์สมรรถนะสูง และเครือข่ายคอมพิวเตอร์ (HPCNC) ที่ช่วยพูดคุยแลกเปลี่ยนความคิด ตลอดจนใช้ช่วงเวลาที่ดีมาด้วยกันตลอดเวลา วันเวลาที่ผ่านมาเป็นช่วงที่ยากลืมเลือนได้

ขอกราบขอบพระคุณคุณพ่อและคุณแม่ ที่ให้การเลี้ยงดูข้าพเจ้าอย่างเต็มที่ ให้อิสระในการเลือกเส้นทางของข้าพเจ้าเสมอมา ข้าพเจ้าจะต้องตอบแทนพระคุณอย่างสุดความสามารถ

ข้าพเจ้าจะดีใจอย่างยิ่งหากงานวิจัยชิ้นนี้เป็นประโยชน์ต่อสังคมในภายหน้า

พิสิษฐ์ มรรคไพสิฐ

ตุลาคม 2557

สารบัญ

หน้า

สารบัญตาราง

สารบัญภาพ

สารบัญภาพ (ต่อ)

ภาพที่ **หน้า**

เฟรมเวิร์กและเทคนิคการคอมไพล์สำหรับการพัฒนาโปรแกรมบนเครื่องจีพียูคลัสเตอร์ด้วยการใช้ตัวชี้แนะคอมไพเลอร์

The Framework and Compilation Techniques for Directive-based GPU Cluster Programming

คำนำ

เครื่องคอมพิวเตอร์ประสิทธิภาพสูงในยุคปัจจุบันมักออกแบบโดยใช้การประมวลผลแบบกระจาย (Distributed Computing) เป็นสถาปัตยกรรมหลัก และใช้หน่วยประมวลผลกลางร่วมกับการเร่งประสิทธิภาพโดยใช้ฮาร์ดแวร์ (Hardware Acceleration) ในการประมวลผลบนแต่ละโหนด (Chavarria-Miranda *et al*., 2012) จีพียูหรือหน่วยประมวลผลกราฟิก (GPU - Graphic Processor Unit) เป็นเทคโนโลยีหนึ่งของตัวเร่งประสิทธิภาพโดยใช้ฮาร์ดแวร์ที่ได้รับความนิยม มาก เนื่องจากมีจำนวนหน่วยประมวลผลเป็นจำนวนมาก ทั้งยังมีราคาที่ถูกและประหยัดพลังงานมากกว่าการใช้หน่วยประมวลผลกลางหลายตัวร่วมกัน ด้วยเหตุนี้เครื่องคอมพิวเตอร์ประสิทธิภาพสูงจำนวนมากจึงอยู่ในรูปของเครื่องจีพียูคลัสเตอร์ (GPU Cluster)

ในขณะที่ความสามารถของฮาร์ดแวร์ถูกพัฒนาไปอย่างรวดเร็ว โมเดลการพัฒนาโปรแกรมบนเครื่องจีพียูคลัสเตอร์ยังคงสร้างความยากลำบากในการพัฒนาให้กับโปรแกรมเมอร์ ผู้พัฒนาโปรแกรม ต้องจัดการกับการส่งข้อมูลระหว่างโหนดบนการประมวลผลแบบกระจายด้วยตนเอง จึงจะได้โปรแกรมที่ยืดหยุ่นและมีประสิทธิภาพ เช่นเดียวกับจีพียูที่โปรแกรมเมอร์จะต้องจัดการกับหน่วยความจำบนจีพียูที่แยกออกจากหน่วยความจำหลักเอง ทำให้เกิดปัญหาที่ตามมาหลายประการคือ โปรแกรมเมอร์จะต้องศึกษาและทำความเข้าใจสถาปัตยกรรมของทั้งการประมวลแบบกระจายและการประมวลผลบนจีพียูที่ซึ่งมีความซับซ้อนสูง การพัฒนาโปรแกรมเป็นไปได้อย่างยากลำบาก เต็มไปด้วยข้อผิดพลาด และสิ้นเปลืองเวลาในการพัฒนา และต้องอาศัยความเข้าใจสถาปัตยกรรมของฮาร์ดแวร์ในเชิงลึกจึงจะสามารถทำการ ปรับปรุงสมรรถนะของ โปรแกรมได้อย่างมีประสิทธิภาพ

วิธีการแก้ปัญหาวิธีหนึ่งคือการใช้ตัวชี้แนะคอมไพเลอร์และซอร์สทูซอร์สคอมไพเลอร์เพื่อช่วยในการพัฒนาโปรแกรมเชิงขนาน ผู้พัฒนาโปรแกรมสามารมุ่งความสนใจไปที่ของขั้นตอนวิธีแบบขนาน (Parallel Algorithm) แทนขั้นตอนการทำงานที่ซับซ้อนซึ่งเป็นผลมาจากสถาปัตยกรรม ในงานวิจัยนี้ผู้วิจัยจะใช้ตัวชี้แนะคอมไพเลอร์มาตรฐานโอเพนเอซีซีซึ่งเป็นมาตรฐานสำหรับการเขียนโปรแกรมเชิงขนานบนจีพียูมาพัฒนาเพิ่มเติม เพื่อให้สนับสนุนการพัฒนาโปรแกรมบนเครื่องจีพียูคลัสเตอร์ รวมไปถึงการออกแบบเทคนิคการคอมไพล์โปรแกรม รันไทม์ไลบรารี่ เพื่อการแปลงโค้ด และเทคนิคการปรับปรุงสมรรถนะอัตโนมัติของคอมไพเลอร์

วัตถุประสงค์

1. เสนอส่วนต่อขยายสำหรับมาตรฐานโอเพนเอซีซีเวอร์ชัน 1.0 ให้สามารถรองรับการอธิบายโปรแกรมเชิงขนานสำหรับทำงานบนเครื่องจีพียูคลัสเตอร์ได้ โดยที่โปรแกรมจะต้อง

1.1 ใช้ภาษาซีในการเขียน และมีลูปซึ่งสามารถทำงานแบบขนานได้

1.2 มีรูปแบบการเข้าถึงข้อมูลแบบปกติ (Regular Access Pattern) และสามารถวิเคราะห์ได้ในเวลาคอมไพล์

2. เสนอและพัฒนาเทคนิคการแปลงโค้ดแบบซอร์สทูซอร์ส จากมาตรฐานโอเพนเอซีซีบางส่วนที่สนับสนุนและส่วนต่อขยายไปยังโค้ดภาษาซีที่สามารถทำงานบนเครื่องจีพียูคลัสเตอร์ได้ ตลอดจนเฟรมเวิร์กสำหรับการคอมไพล์และรันโปรแกรม

3. เสนอและพัฒนาเทคนิคการปรับสมรรถนะอัตโนมัติ สำหรับซอร์สทูซอร์สคอมไพเลอร์ที่นำเสนอ

การตรวจเอกสาร

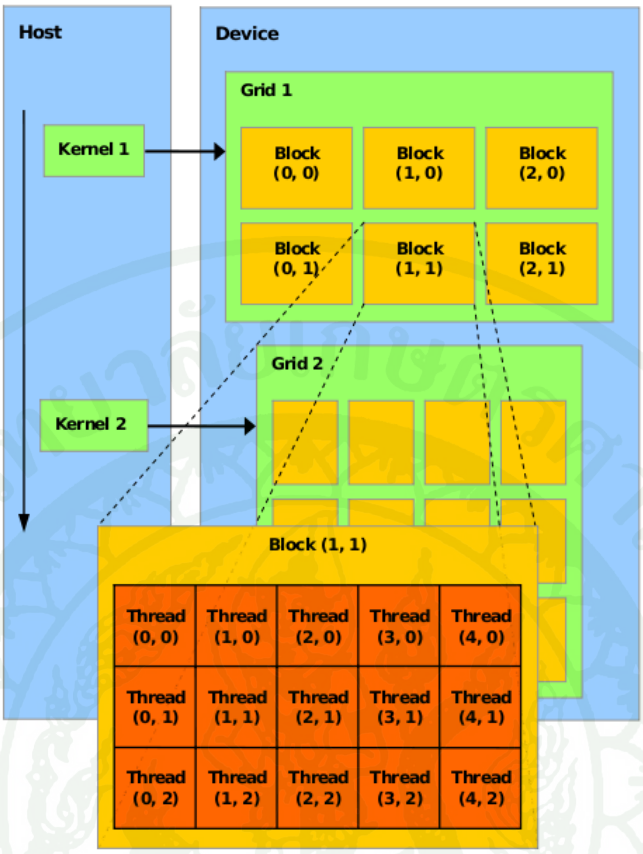
ในหัวข้อนี้จะกล่าวถึงความรู้พื้นฐาน ที่จำเป็น และงานวิจัยอื่นที่เกี่ยวข้องหรือเป็นแหล่งอ้างอิงของงานวิจัยนี้ ความรู้เบื้องต้นที่สำคัญได้แก่ การประมวลโปรแกรมทั่วไปด้วยจีพียูหรือจีพีจีพียู (GPGPU - General-Purpose computation on Graphics Processing Units) โอเพนซีแอล (OpenCL) และไลบรารีมาตรฐานสำหรับการพัฒนาโปรแกรมแบบส่งผ่านข้อความ (Message-passing Programming) หรือ เอ็มพีไอ (MPI)

จีพีจีพียูและโอเพนซีแอล

หน่วยประมวลผลกราฟิกหรือจีพียูมีการใช้งานมายาวนานในด้านการประมวลผลกราฟิกให้กับคอมพิวเตอร์ และมีการพัฒนาอย่างต่อเนื่องในด้านความสามารถในการประมวลผลและประสิทธิภาพในการใช้พลังงาน ทำให้เกิดแนวคิดในการนำจีพียูมาใช้กับการประมวลผลทั่วไปด้วย ซึ่งเรียกแนวคิดนี้ว่า จีพีจีพียู ผู้ผลิตกราฟิกการ์ดเองก็เริ่มมีการพัฒนาเครื่องมือและภาษาสำหรับช่วยให้นักพัฒนาโปรแกรมพัฒนาโปรแกรมที่ทำงานทั่วไปบนจีพียูได้

ในปีค.ศ. 2007 เอ็นวิเดีย (NVIDIA) ผู้ผลิตกราฟิกการ์ดรายใหญ่ได้เสนอสถาปัตยกรรมคูดา (CUDA) ออกมาให้ผู้พัฒนาโปรแกรมใช้งาน คูดาเป็นสถาปัตยกรรมที่ออกแบบขึ้นมาเพื่อรองรับการประมวลผลแบบจีพีจีพียู ซึ่งประกอบด้วย

1. ชุดคำสั่ง (ISA) ของจีพียู
2. สถาปัตยกรรมการประมวลผลในจีพียู มีรูปแบบเป็นการประมวลผลแบบขนาน และมีหน่วยความจำแบบลำดับชั้น (Hierarchical Memory)
3. ภาษาระดับสูงที่เป็นส่วนขยายของภาษาซีสำหรับใช้ในการพัฒนาโปรแกรม



ภาพที่ 1 สถาปัตยกรรมคูดา

โมเดลการประมวลผลของคูดาจะเป็นแบบซิมดี (SIMD) ซึ่งแบ่งเทรด (Thread) ทั้งหมดออกเป็นกลุ่มย่อยที่เรียกว่าเทรดบล็อค (Thread Block) โดยเทรดทั้งหมดที่อยู่ในเทรดบล็อคเดียวกันจะถูกประมวลผลแบบเสมือนพร้อมกันบนสตรีมมัลติโปรเซสเซอร์ (Stream Multiprocessor หรือ SM) เดียวกัน และสามารถใช้งานหน่วยความจำแชร์ (Shared Memory) ร่วมกันได้ ดังภาพที่ 1 สถาปัตยกรรมคูดา ในส่วนนี้โปรแกรมเมอร์ยังคงที่จะต้องกำหนดขนาดของเทรดบล็อคด้วยตัวเองเช่นกัน จึงกล่าวได้ว่าโมเดลการพัฒนาโปรแกรมของคูดาที่นำเสนอขึ้นมานั้น ยังคงมีความเป็นภาษาระดับต่ำที่โปรแกรมเมอร์จะต้องยุ่งเกี่ยวกับการจัดการฮาร์ดแวร์ด้วยตนเอง ส่งผลให้พัฒนาโปรแกรมได้ล่าช้าและเสี่ยงต่อการสร้างข้อผิดพลาด

คูดายังมีข้อจำกัดที่สำคัญอีกประการหนึ่ง นั่นคือสามารถใช้ได้กับเฉพาะจีพียูที่บริษัทเอ็นวิเดียเป็นผู้ผลิตเท่านั้น มาตรฐานโอเพนซีแอลจึงถูกเสนอขึ้นเพื่อให้โปรแกรมเมอร์สามารถพัฒนาโปรแกรมให้ใช้งานกับจีพียูของผู้ผลิตรายอื่นได้ และยังขยายไปถึง ตัวเร่งฮาร์ดแวร์ ประเภท

อื่น โอเพนซีแอลถูกออกแบบมาให้เป็นไลบรารีภายนอก ทำให้ไม่ต้องการคอมไพเลอร์ที่สร้างขึ้นเฉพาะ สามารถติดตั้งไลบรารี่และใช้งานได้ทันที มีฟังก์ชันสำหรับสนับสนุนการทำงานในหลายแพลตฟอร์ม และหลายตัวเร่งฮาร์ดแวร์ในเครื่องเดียว

การพัฒนาโปรแกรมแบบส่งผ่านข้อความและเอ็มพีไอ

การพัฒนาโปรแกรมแบบส่งผ่านข้อความ เป็นการพัฒนาโปรแกรมเชิงขนานรูปแบบหนึ่ง ซึ่งมักจะใช้กับสถาปัตยกรรมแบบ NUMA (Non-uniform memory access) ที่มีหน่วยความจำเป็นแบบกระจาย (Distributed Memory) ซึ่งทำให้แต่ละหน่วยประมวลผลไม่สามารถติดต่อสื่อสารกันได้ผ่านทางหน่วยความจำร่วม จึงทำให้เกิดการสื่อสารผ่านการส่งข้อมูลหากันระหว่างเครื่องผ่านระบบเน็ตเวิร์ก

เอ็มพีไอเป็นไลบรารีมาตรฐานของการพัฒนาโปรแกรมแบบส่งผ่านข้อความที่พัฒนาต่อยอดมาจากพีวีเอ็ม (PVM - Parallel Virtual Machine) ใช้งานได้กับภาษา C, C++, Fortran และ Java ถูกนำมาใช้กับการพัฒนาโปรแกรมบนระบบคลัสเตอร์คอมพิวเตอร์มายาวนาน และได้รับการพัฒนามาอย่างต่อเนื่อง การพัฒนาโปรแกรมแบบส่งผ่านข้อความ ผู้พัฒนาจะต้องมีการกำหนดการส่งข้อมูลด้วยตนเอง ว่าจะให้โพรเซสใดเป็นผู้ส่งหรือผู้รับและส่งข้อมูลส่วนใดบ้าง ต้องมีการกำกับจังหวะการทำงานให้ถูกต้อง มิเช่นนั้นโปรแกรมอาจจะทำงานต่อไม่ได้ หรือทำงานผิดพลาด

ในการติดต่อสื่อสารระหว่างเครื่องด้วยเอ็มพีไอ จะมีฟังก์ชันการทำงานอยู่สองกลุ่ม ได้แก่ การติดต่อสื่อสารระหว่างเครื่องสองเครื่อง (Point to Point Communication) และ การติดต่อสื่อสารระหว่างกลุ่มเครื่อง (Collective Communication)

การติดต่อสื่อสารระหว่างเครื่องสองเครื่องเป็นรูปแบบการติดต่อสื่อสารที่มีความหมายตามชื่อ นั่นคือสำหรับการติดต่อหากันระหว่างสองเครื่องเท่านั้น ฟังก์ชันพื้นฐานในกลุ่มนี้ ได้แก่

1. MPI_Send – ฟังก์ชันสำหรับส่งข้อมูลไปยังอีกโพรเซสหนึ่ง
2. MPI_Recv – ฟังก์ชันสำหรับรับข้อมูลจากอีกโพรเซสหนึ่ง

การติดต่อสื่อสารระหว่างกลุ่มเครื่องเป็นรูปแบบที่มีการส่งข้อมูลระหว่างกลุ่มของโพรเซส เช่น การส่งข้อมูลจากโพรเซสหนึ่งไปยังโพสเซสอื่นทั้งหมด หรือการรวมข้อมูลจากทุกโพรเซสมาไว้ในโพรเซสเดียวกัน เป็นต้น ฟังก์ชันการทำงานเหล่านี้แม้จะสามารถเขียนขึ้นเองด้วยฟังก์ชันการติดต่อสื่อสารระหว่างเครื่องสองเครื่อง แต่ก็ยากที่จะทำให้มีประสิทธิภาพ การใช้ฟังก์ชันที่ไลบรารีเตรียมไว้จะช่วยให้ดึงประสิทธิภาพออกมาได้มากกว่า ฟังก์ชันที่สำคัญในกลุ่มนี้ ได้แก่

1. MPI_Bcast – ฟังก์ชันสำหรับการกระจายข้อมูลแบบบรอดคาซท์ (Broadcast) สำหรับกระจายข้อมูลจากโพรเซสหนึ่งไปยังโพรเซสอื่นทั้งหมด โดยที่ข้อมูลที่ส่งไปยังทุกโพรเซสจะเป็นข้อมูลเดียวกัน

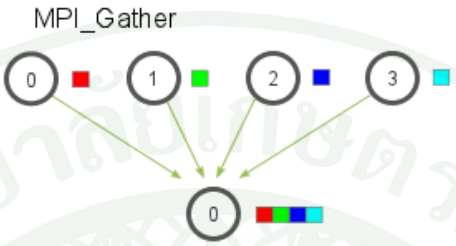
2. MPI_Reduce – ฟังก์ชันสำหรับการรีดิวซ์ข้อมูล (Reduction) สำหรับดำเนินการกับข้อมูลหนึ่งของทุกโปรเซสให้เหลือเพียงตัวเดียว เช่น การหาผลรวมของการบวก การหาผลรวมของการคูณ หรือการหาค่ามากที่สุด เป็นต้น รีดักชันถือเป็นโอเปอเรเตอร์ที่สำคัญอีกตัวหนึ่งของการโปรแกรมเชิงขนาน

3. MPI_Allreduce – ฟังก์ชันสำหรับการรีดิวซ์ข้อมูล มีข้อแตกต่างจาก MPI_Reduce คือ เมื่อสิ้นสุดการทำงานของ MPI_Reduce ค่าที่รีดิวซ์จะอยู่ที่รูทโพรเซสที่กำหนดไว้เท่านั้น โพรเซสอื่นจะไม่ได้ค่ารีดิวซ์ แต่ MPI_Allreduce จะให้ค่ารีดิวซ์กับทุกโพรเซสหลังการทำงานเสร็จสิ้น

4. MPI_Scatter – ฟังก์ชันสำหรับการกระจายข้อมูลแบบสแคทเทอร์ (Scatter) สำหรับการกระจายข้อมูลจากโพรเซสหนึ่งไปยังโพรเซสอื่นทั้งหมด ข้อมูลที่กระจายไม่จำเป็นต้องเป็นข้อมูลเดียวกัน สามารถเป็นข้อมูลชุดเดียวกันและแบ่งออกไปตามแต่ละโพรเซสได้ ความแตกต่างระหว่างการทำงานของ MPI_Bcast และ MPI_Scatter สามารถดูได้จากภาพที่ 2

ภาพที่ 2 ความแตกต่างของทำงานระหว่าง MPI_Bcast และ MPI_Scatter

5. MPI_Gather – ฟังก์ชันสำหรับการรวมข้อมูลแบบแกทเธอร์ (Gather) สำหรับการรวมข้อมูลจากโพรเซสอื่นมาไว้ยังโพรเซสหนึ่ง เป็นฟังก์ชันที่มีการทำงานตรงกันข้ามกับ MPI_Scatter ตัวอย่างการทำงานของฟังก์ชันนี้เป็นดังภาพที่ 3



ภาพที่ 3 การรวบรวมข้อมูลแบบแกทเธอร์

6. MPI_Barrier – ฟังก์ชันสำหรับประสานจังหวะ (Synchronization) ใช้รอทุกโพรเซสที่ทำงานอยู่ให้มาถึงจุดของ MPI_Barrier แล้วจึงเริ่มทำงานพร้อมกันอีกครั้ง

การติดต่อสื่อสารของเอ็มพีไอ ยังสามารถแบ่งออกเป็นการสื่อสารแบบกีดขวางการรับส่งข้อมูล (Blocking Communication) และการสื่อสารแบบไม่กีดขวางการรับส่งข้อมูล (Non-blocking Communication) การสื่อสารแบบกีดขวางการรับส่งข้อมูลคือการสื่อสารแบบที่ฟังก์ชันของการรับหรือการส่ง จะยังทำงานไม่จบหากยังรับส่งข้อมูลไม่เสร็จ ฟังก์ชันพื้นฐานที่กล่าวมาก่อนหน้านี้ทั้งหมดจะอยู่ในกลุ่มนี้ ซึ่งตรงกันข้าม การสื่อสารแบบไม่กีดขวางการรับส่งข้อมูลคือการที่โปรแกรมสามารถเริ่มทำงานคำสั่งอื่นต่อได้ทันที ถึงแม้จะยังรับส่งข้อมูลไม่เสร็จ ซึ่งจะช่วยแก้ปัญหาในเรื่องการติดตาย (Deadlock) ในบางกรณี และเกิดการทำงานแบบขนานมากขึ้น ฟังก์ชันที่มีการสื่อสารระหว่างเครื่องสองเครื่องแบบไม่กีดขวางการรับส่งข้อมูล ได้แก่ MPI_Isend และ MPI_Irecv ซึ่งมีการทำงานเหมือนกับ MPI_Send และ MPI_Recv แต่ไม่ต้องรอให้ทำงานเสร็จก็สามารถทำงานคำสั่งอื่นไปพร้อมกันได้

ฟังก์ชันที่มีการสื่อสารระหว่างกลุ่มแบบไม่กีดขวางการรับส่งข้อมูล ได้แก่ MPI_Ibcast MPI_Ireduce MPI_Iscatter และ MPI_Igather เป็นต้น ทุกฟังก์ชันที่ไม่มีการกีดขวางการรับส่งข้อมูล จะต้องเรียกฟังก์ชันสำหรับกีดขวางการทำงาน MPI_Wait เพื่อให้โปรแกรมรอการรับส่งข้อมูลจนเสร็จสิ้น

สำหรับฟังก์ชันที่มีการสื่อสารระหว่างกลุ่มแบบไม่กีดขวางการรับส่งข้อมูล จะเริ่มมีในมาตรฐานเอ็มพีไอเวอร์ชันสาม (MPI-3) เป็นต้นไปเท่านั้น ในเวอร์ชันก่อนหน้าจะยังไม่สามารถใช้ได้ อิมพลีเมนเตชันของมาตรฐานเอ็มพีไอ มีการสร้างขึ้นมาจากหลายกลุ่ม เช่น MPICH LAM/MPI และ OpenMPI เป็นต้น ในการเลือกใช้งานจะต้องระวังในเรื่องของมาตรฐานของเอ็มพีไอ เนื่องจากแต่ละเวอร์ชันของแต่ละอิมพลีเมนเตชันจะใช้เวอร์ชันของมาตรฐานเอ็มพีไอไม่ตรงกัน

โมเดลการพัฒนาโปรแกรมโดยใช้ตัวชี้แนะคอมไพเลอร์บนจีพียู

วิธีการพัฒนาโปรแกรมโดยใช้ตัวชี้แนะคอมไพเลอร์เป็นอีกวิธีที่มีประสิทธิภาพในการซ่อนรายละเอียดของสถาปัตยกรรมระดับต่ำของระบบจากขั้นตอนวิธีแบบขนาน ด้วยการพัฒนาแบบทำซ้ำ (Iterative method) ผู้พัฒนาโปรแกรมสามารถทำให้โปรแกรมเชิงลำดับกลายเป็นโปรแกรมเชิงขนานด้วยการเพิ่มตัวชี้แนะคอมไพเลอร์เข้าไปที่โค้ดของโปรแกรม

การเขียนตัวชี้แนะคอมไพเลอร์ประกอบด้วยสองส่วนคือ แพร็กมา (Pragma หรือ Construct) เป็นส่วนที่ใช้กำหนดรูปแบบการทำงานในโปรแกรม และ คลอส (Clause) ซึ่งเป็นการระบุคุณสมบัติการทำงานเพิ่มเติมให้กับแต่ละแพร็กมา ตัวอย่างเช่น ตัวชี้แนะคอมไพเลอร์ของโอเพนเอ็มพี (OpenMP) สำหรับการระบุให้มีการทำงานแบบขนานในระดับเทรด ด้วยจำนวนเทรดเท่ากับ 4 เทรด จะเขียนด้วย #pragma omp parallel num_threads(4) โดยที่คำว่า parallel คือแพร็กมาที่เป็นคำสั่งให้คำสั่งหลังจากนี้ทำงานแบบขนาน และ num_threads(4) เป็นคลอสที่ระบุจำนวนเทรด ซึ่งในที่นี้มีพารามิเตอร์เป็น 4 ซึ่งหมายถึงระบุให้มีจำนวนเทรดเท่ากับ 4

จากความสำเร็จของโอเพนเอ็มพี (OpenMP) ที่แก้ปัญหาการเขียนโปรแกรมเชิงขนานแบบใช้หน่วยความจำร่วม (Shared Memory) ด้วยการนำตัวชี้แนะคอมไพเลอร์เข้ามาใช้ โดยให้โปรแกรมเมอร์กำกับส่วนของโปรแกรมที่ต้องการให้ประมวลผลแบบขนาน คุณลักษณะของข้อมูล การประสานจังหวะระหว่างเทรด และรูปแบบการกระจายงาน ทำให้นักวิจัยหลายกลุ่มเกิดแนวคิดที่จะนำวิธีการนี้มาใช้เป็นโมเดลสำหรับการพัฒนาโปรแกรมบนจีพียู นักวิจัยกลุ่มแรกๆที่เริ่มต้นศึกษาคือ Lee *et al.*, (2009) โดยนำตัวชี้แนะคอมไพเลอร์ของโอเพนเอ็มพีมาใช้สำหรับการพัฒนาโปรแกรมบนจีพียูโดยตรง ด้วยการใช้ซอร์สทูซอร์สคอมไพเลอร์สำหรับการแปลงโปรแกรมภาษาซีที่มีตัวชี้แนะคอมไพเลอร์ของโอเพนเอ็มพี ไปยังภาษาคูดา งานวิจัยชิ้นนี้แสดงให้เห็นว่าแต่

ละรอบการทำงานของลูปที่ถูกกำกับด้วย omp for ของโอเพนเอ็มพีสามารถแปลงให้เป็นการทำงานของแต่ละเทรดบนจีพียูได้ คำสั่งประสานจังหวะของโอเพนเอ็มพีก็สามารถแปลงเป็นคำสั่งประสานจังหวะของคูดาได้โดยตรง เพียงแต่ต้องมีการสร้างจุดแบ่ง (Split Point) ขึ้นมาและแบ่งโค้ดออกจากกันเพื่อให้โปรแกรมทำงานได้ถูกต้องบนจีพียู จุดที่เป็นปัญหาคือตัวชี้แนะคอมไพเลอร์ของโอเพนเอ็มพีถูกสร้างขึ้นโดยมีจุดประสงค์เพื่อใช้กับโปรแกรมเชิงขนานแบบใช้หน่วยความจำร่วมเป็นหลัก จึงยังไม่สามารถดึงความสามารถของจีพียูได้อย่างเต็มที่ นอกจากนี้ในส่วนของการจัดการข้อมูลบนจีพียูยังมีอีกหลายปัญหาที่ยังไม่สามารถแก้ได้หรือทำได้ไม่ดีเช่น การเข้าใช้งานหน่วยความจำโกลบอล (Global Memory) อย่างมีประสิทธิภาพ การพยายามใช้ประโยชน์ จากหน่วยความจำแชร์ (Shared Memory) เป็นต้น

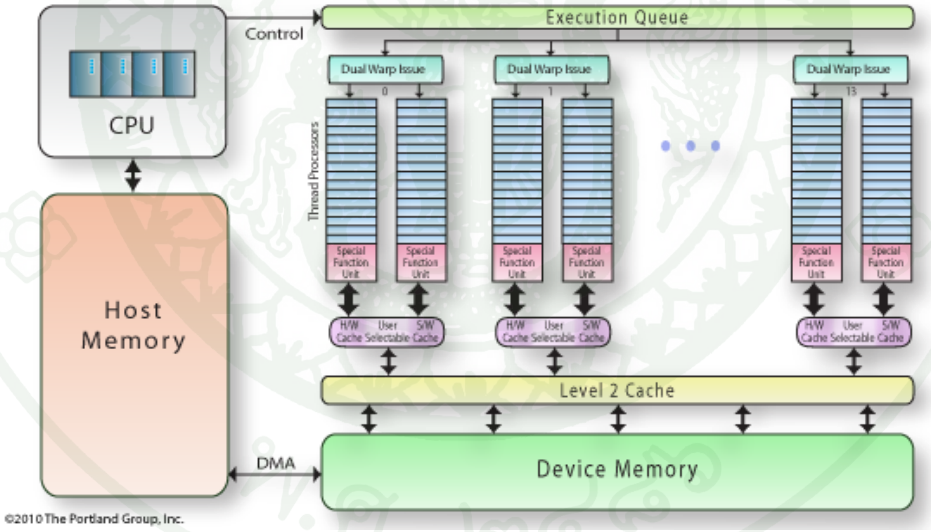
Lee *et al*., (2010) ได้ทำการศึกษาและปรับปรุงงานวิจัยชิ้นเดิมจนได้ผลลัพท์ออกมาเป็นโอเพนเอ็มพีซี (OpenMPC) ตัวชี้แนะคอมไพเลอร์ที่ใช้ในงานวิจัยใหม่นี้ยังคงมีพื้นฐานมาจากโอเพนเอ็มพี แต่ได้เพิ่มเติมในส่วนของพารามิเตอร์ให้หลากหลายขึ้น ซึ่งจะทำให้คอมไพเลอร์สามารถรับข้อมูลคุณลักษณะต่างๆของโปรแกรมเพิ่มเติม เพื่อนำไป ปรับปรุงสมรรถนะ โปรแกรมได้ดียิ่งขึ้น เพิ่มการวิเคราะห์ข้อมูลเพื่อนำไปใช้ช่วยในการ ปรับปรุงสมรรถนะ หน่วยความจำบนจีพียู และด้วยการที่มีพารามิเตอร์จำนวนมากให้ทดสอบ งานชิ้นนี้ยังมีส่วนของการปรับปรุง (Tuning) ระบบให้เหมาะสมที่สุดด้วยการค้นหาค่าพารามิเตอร์ที่ดีที่สุดอัตโนมัติโดยใช้การค้นหาในปริภูมิ (Search Space) ส่งผลให้ประสิทธิภาพโดยรวมของโปรแกรมที่ได้ดีขึ้นกว่างานวิจัยเดิม

พีจีไอแอคเซราเรเตอร์ (PGI Accelerator) เป็นตัวชี้แนะคอมไพเลอร์ที่พัฒนาขึ้นมาจากภาคธุรกิจ โดยถูกคิดค้นขึ้นจากบริษัทพีจีไอ (PGI) ผู้ผลิตคอมไพเลอร์สำหรับเครื่องคอมพิวเตอร์ประสิทธิภาพสูง พีจีไอแอคเซราเรเตอร์จะไม่มุ่งเป้าไปที่การสร้างตัวชี้แนะคอมไพเลอร์สำหรับจีพียูอย่างเดียว แต่จะออกแบบเพื่อให้รองรับตัวเร่งฮาร์ดแวร์ อื่นด้วย และพยายามออกแบบมาให้โปรแกรมเมอร์ไม่จำเป็นต้องมีการแก้โค้ดต้นฉบับ (ยกเว้นในส่วนของการเพิ่มตัวชี้แนะคอมไพเลอร์) โดยนำเสนอตัวชี้แนะคอมไพเลอร์ที่ประกอบด้วยคำสั่งประเภทหลัก ได้แก่ 1) ส่วนกำหนดการทำงานเชิงขนาน และ 2) ส่วนจัดการกับข้อมูล

ส่วนกำหนดการทำงานเชิงขนานของพีจีไอแอคเซราเรเตอร์จะมีคำสั่งสำคัญคือ region ซึ่งจะใช้บอกชุดของคำสั่งที่ต้องการย้ายขึ้นไปทำงานบนตัวเร่งฮาร์ดแวร์ ดังนั้นโปรแกรมเมอร์จึงสามารถพัฒนาโปรแกรมได้ง่าย แต่จะไม่สามารถทำการปรับปรุงสมรรถนะด้วยตนเองได้ เนื่องจาก

ตัวเร่งฮาร์ดแวร์ หลักที่นิยมใช้กันในปัจจุบันคือจีพียู ที่ซึ่งสถาปัตยกรรมของหน่วยความจำมีความซับซ้อนสูง ส่วนจัดการกับข้อมูลจึงมีคำสั่งเป็นจำนวนมาก มีความยุ่งยากสูง เพื่อแลกกับประสิทธิภาพการทำงาน ข้อด้อยอีกประการของพีจีไอแอคเซราเรเตอร์คือ ไม่มีคำสั่ง reduction คอมไพเลอร์จะเป็นผู้วิเคราะห์เองว่าตัวแปรตัวใดที่จะทำการรีดักชัน ซึ่งถ้าหากเป็นโค้ดที่ซับซ้อนก็มีโอกาสที่คอมไพเลอร์จะหารูปแบบที่ถูกต้องไม่ได้ส่งผลให้ทำงานผิดพลาด (Lee *et al*., 2012; Reyes *et al*., 2012)

ตัวชี้แนะคอมไพเลอร์ได้ถูกนำเสนอขึ้นมามากมายสำหรับการพัฒนาโปรแกรมบนจีพียูหรือตัวเร่งฮาร์ดแวร์ แต่ยังไม่มีมาตรฐานที่แน่นอน ทำให้เกิดความร่วมมือของสี่บริษัทใหญ่ได้แก่ แคปส์ (CAPS) เครย์ (CRAY) พีจีไอ และเอ็นวิเดีย เพื่อสร้างและผลักดันตัวชี้แนะคอมไพเลอร์มาตรฐานร่วมกันขึ้น จึงเป็นที่มาของโอเพนเอซีซี (OpenACC) ซึ่งเป็นมาตรฐานที่นิยมใช้กันแพร่หลายที่สุดในปัจจุบัน



ภาพที่ 4 โมเดลการทำงานของโอเพนเอซีซี

โอเพนเอซีซีจะมีโมเดลการทำงานเป็นไปตาม ภาพที่ 4 โดยมีสมมติฐานคือเครื่องโฮสและดีไวซ์ จะมีหน่วยความจำแยกจากกัน หน่วยประมวลผลกลางของโฮสไม่สามารถเข้าถึงหน่วยความจำของดีไวซ์ได้โดยตรง ขณะที่หน่วยประมวลผลของดีไวซ์ก็ไม่สามารถ เข้าถึงหน่วยความจำของโฮสได้โดยตรงเช่นกัน ทั้งหมดนี้ต้องผ่านการส่งข้อมูลหากันระหว่างหน่วยความจำของโฮสและดีไวซ์ผ่านดีเอ็มเอ (DMA หรือ Direct Memory Access)

ภาพที่ 5 เป็นตัวอย่างการใช้โอเพนเอซีซีกับโปรแกรม Jacobi Relaxation แต่ละรอบของลูป for ที่อยู่ด้านนอกสุด สามารถทำงานพร้อมกันได้ ดังนั้นจึงใช้แพร็กมา parallel loop กำกับไว้เพื่อระบุว่าลูปนี้สามารถทำงานแบบขนานได้ คลอส create ใช้ระบุว่าตัวแปร newa ไม่จำเป็นต้องใช้การส่งข้อมูลระหว่างโฮสกับดีไวซ์ ให้สร้างตัวแปรที่ทำงานเฉพาะบนดีไวซ์อย่างเดียว คลอส copy ระบุว่าตัว a จะต้องมีการส่งจากโฮสไปยัง ดีไวซ์ก่อนเริ่มประมวลผลเคอร์เนล และต้องส่งจาก ดีไวซ์กลับมายังโฮสเมื่อประมวลผลเสร็จ สุดท้ายคลอส reduction กำหนดให้ตัวแปร change ของลูปแต่ละรอบจะต้องถูกรีดิวซ์เหลือเพียงค่าเดียวด้วยโอเปอเรเตอร์ max

```
do{
  change = 0;
  #pragma acc parallel loop create(newa[0:n][0:m]) \
     copy(a[0:n][0:m]) reduction(max:change)
  for( j = 1; j < m-1; ++j ){
    for( i = 1; i < n-1; ++i ){
      newa[j][i] = w0*a[j][i] +
         w1 * (a[j][i-1] + a[j-1][i] + a[j][i+1] + a[j+1][i]) +
         w2 * (a[j-1][i-1] + a[j+1][i-1] + a[j-1][i+1] +
a[j+1][i+1]);
         change = fmax(change,fabs(newa[j][i]-a[j][i]));
    }
  }
  tmp = a; a = newa; newa = tmp;
}while( change > tolerance );
```

ภาพที่ 5 ตัวอย่างการใช้แพร็กมา parallel loop ในโปรแกรม Jacobi Relaxation

โปรแกรมตัวอย่างในภาพที่ 5 สามารถทำงานได้อย่างถูกต้อง แต่จะใช้เวลาประมวลผลนาน เนื่องจากด้านนอกสุดถูกครอบไว้ด้วยลูป do while ซึ่งหมายถึงว่าจำนวนการส่งข้อมูลระหว่างโฮสและดีไวซ์จะเท่ากับจำนวนรอบของลูป do while ซึ่งที่จริงแล้วการส่งข้อมูลนั้นสามารถส่งไปและส่งกลับอย่างละครั้งก่อนและหลังลูปก็เพียงพอที่จะทำให้โปรแกรมทำงานถูกต้อง ใน ภาพที่ 6 แพร็กมา data ถูกใช้เพื่อให้โปรแกรมทำการส่งข้อมูลไปกลับเพียงครั้งเดียว ในช่วงก่อนและหลังการทำงานของลูป do while

```
#pragma acc data create(newa[0:n][0:m]) copy(a[0:n][0:m])
do{
  change = 0;
  #pragma acc parallel loop reduction(max:change)
  for( j = 1; j < m-1; ++j ){
    for( i = 1; i < n-1; ++i ){
      newa[j][i] = w0*a[j][i] +
        w1 * (a[j][i-1] + a[j-1][i] + a[j][i+1] + a[j+1][i]) +
        w2 * (a[j-1][i-1] + a[j+1][i-1] + a[j-1][i+1] +
a[j+1][i+1]);
        change = fmax(change,fabs(newa[j][i]-a[j][i]));
    }
  }
  tmp = a; a = newa; newa = tmp;
}while( change > tolerance );
```

ภาพที่ 6 ตัวอย่างการใช้แพร็กมา data ในโปรแกรม Jacobi Relaxation

เนื่องจากโอเพนเอซีซีเป็นมาตรฐานของตัวชี้แนะคอมไพเลอร์สำหรับจีพียู และมีความสามารถใช้การปรับใช้กับโปรแกรมเชิงขนานจำนวนมาก งานวิจัยนี้จึงเลือกที่จะใช้มาตรฐานโอเพนเอซีซีสำหรับนำมาพัฒนาต่อยอด

โมเดลการพัฒนาโปรแกรมโดยใช้ตัวชี้แนะคอมไพเลอร์บนจีพียูคลัสเตอร์

ตัวชี้แนะคอมไพเลอร์ยังคงมีส่วนสำคัญในการเข้ามาช่วยแก้ปัญหาในการประมวลผลแบบกระจาย Basumallik *et al*., (2005) ได้สร้างซอร์สทูซอร์สคอมไพเลอร์ที่แปลงจากโค้ดที่ใช้ตัวชี้แนะคอมไพเลอร์โอเพนเอ็มพี เป็นโค้ดโปรแกรมที่ใช้งานคำสั่งของเอ็มพีไอ ซึ่งในขั้นตอนการแปลงโค้ดนั้น คำสั่ง omp for สามารถเปลี่ยนให้เป็นโค้ดที่แบ่งงานไปยังแต่ละโปรเซสได้โดยตรง คำสั่ง omp section สามารถแทนด้วยคำสั่ง switch-case เพื่อเลือกให้แต่ละโปรเซสประมวลผล คำสั่งประสานจังหวะ (เช่น omp barrier, omp critical, omp flush ฯลฯ) จะถูกจัดการด้วยการใช้กราฟควบคุมสายงาน (Control Flow Graph) และคำสั่ง reduction จะสามารถแทนด้วยการเรียกฟังก์ชัน MPI_Allreduce ของโอเพนเอ็มพี ข้อดีของวิธีการนี้คือสามารถใช้งานได้กับทั้งโปรแกรมที่มีการเข้าถึงข้อมูลทั้งแบบปกติและไม่ปกติ (Irregular Access Pattern) แต่ยากที่จะนำมาปรับใช้กับระบบจีพียูคลัสเตอร์ เพราะโปรแกรมประยุกต์ที่สามารถทำงานได้ดีบนจีพียู ถูกจำกัดด้วยสถาปัตยกรรมของจีพียูที่มีการแบ่งหน่วยความจำของโฮสและดีไวซ์ การส่งข้อมูลที่จำเป็นในช่วงที่หน่วยประมวลผลต้องการจึงทำให้การทำงานไม่มีประสิทธิภาพในด้านเวลาการประมวลผล

โอเอ็มพีเอสเอส (OmpSs) (Bueno *et al*., 2012) เป็นโมเดลการพัฒนาโปรแกรมเชิงขนานที่ตัวชี้แนะคอมไพเลอร์มีพื้นฐานมาจากโอเพนเอ็มพี สามารถรองรับการเขียนโปรแกรมได้ทั้งบนเครื่องจีพียูคลัสเตอร์ และเครื่องแบบหลายจีพียูหรือมัลติจีพียู (Multi-GPU) งานวิจัยนี้แตกต่างจากงานวิจัยที่ผ่านมาเป็นอย่างมาก เนื่องจากขณะที่งานวิจัยอื่นจะเน้นไปที่ การประมวลผลแบบขนานเชิงข้อมูล (Data Parallelism) ซึ่งมักจะมุ่งเป้าไปที่คำสั่งลูป for เป็นหลัก ผู้พัฒนาโปรแกรมจะต้องกำหนดตัวชี้แนะคอมไพเลอร์เพื่อกำหนดรูปแบบ การประมวลผลแบบขนานเชิงการทำงาน (Task Parallelism) ซึ่งจะมุ่งเป้าไปที่คำสั่ง task บนโอเพนเอ็มพี แต่ในส่วนของจีพียูเคอร์เนลต้องเขียนด้วยตนเองด้วยคูดาหรือโอเพนซีแอล

การปรับปรุงสมรรถนะโปรแกรมบนจีพียูคลัสเตอร์

การลดเวลาในการเข้าถึงหน่วยความจำ และการเพิ่มการทำงานพร้อมกันระหว่างการประมวลผลเคอร์เนลและการส่งข้อมูลในหน่วยความจำ เป็นแนวทางหลักสำคัญในการเพิ่มประสิทธิภาพของโปรแกรมที่ทำงานบนจีพียู วิธีการในการลดเวลาในการเข้าถึงหน่วยความจำเพื่อลดเวลาการทำงานรวม เช่น การเพิ่มโลคอลลิตี (Locality) ในการทำงาน การเลี่ยงการใช้หน่วยความจำโกลบอล และใช้หน่วยความจำแชร์ที่มีความเร็วในการเข้าถึงสูงกว่าแทนเป็นต้น

การทำให้เกิดการทำงานพร้อมกัน (Overlapping) เป็นเทคนิคที่มีผลต่อความเร็วในการประมวลผลของจีพียูมาก เนื่องจากการเร่งการประมวลผลด้วยการใช้ ตัวเร่งฮาร์ดแวร์ จะมีจุดอ่อนอยู่ที่เวลาที่เพิ่มขึ้นในขณะส่งข้อมูล ดังนั้นการลดเวลาส่วนนี้ด้วยการทำให้เกิดการทำงานพร้อมกันจึงจำเป็นอย่างยิ่ง ซึ่งรูปแบบทั่วไปในการทำงานพร้อมกัน ได้แก่

1. การทำให้เกิดการพร้อมกันระหว่างการส่งข้อมูล (Overlapping Data Trasfer) เช่น การส่งข้อมูลบางส่วนจากจีพียูไปยังโฮส และข้อมูลบางส่วนจากโฮสไปยังจีพียูสามารถทำงานพร้อมกันได้ ซึ่งในคูดาจะจะใช้สตรีมในการกำหนดลำดับการทำงานและการทำงานพร้อมกัน และในโอเพนซีแอลก็จะมีคิวและอีเวนต์สำหรับการทำให้เกิดการทำงานพร้อมกันของการส่งข้อมูลได้

2. การทำให้เกิดการพร้อมกันระหว่างการส่งข้อมูลและเคอร์เนล (Overlapping Data Transfer and GPU Kernel) เช่น การส่งข้อมูลจากโฮสไปจีพียูหรือจากจีพียูไปโฮส สามารถทำพร้อมกับการประมวลผลเคอร์เนลได้ ถ้าไม่มีการขึ้นต่อกันของข้อมูล

อุปกรณ์และวิธีการ

อุปกรณ์

เพื่อให้สามารถทดสอบการทำงานของโปรแกรมที่ได้จากเฟรมเวิร์กที่นำเสนอ จึงจำเป็นต้องมีเครื่องจีพียูคลัสเตอร์สำหรับทดสอบการทำงานและเวลาในการประมวลผล ในงานวิจัยนี้จะใช้เครื่องเมฆา (maeka) ซึ่งเป็นจีพียูคลัสเตอร์ขนาด 8 โหนด เชื่อมต่อกันด้วย กิกะบิตอีเธอร์เน็ต (Gigabit Ethernet) และมีรายละเอียดทางเทคนิคของแต่ละเครื่องเป็นดังนี้

1. CPU Intel Xeon E5620 @ 2.40GHz
2. NVIDIA Tesla M2050 (14 MP, 32 Cores/MP and 1.15 GHz)

ซอฟต์แวร์ที่ใช้ได้แก่

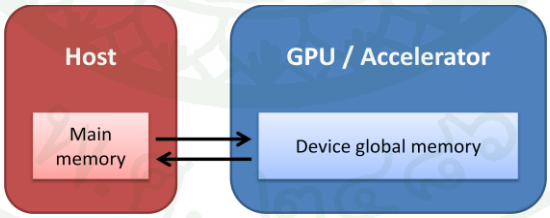
1. CUDA runtime เวอร์ชัน 4.2.1
2. OpenCL เวอร์ชัน 1.1
3. คอมไพเลอร์ gcc เวอร์ชัน 4.1.2 สำหรับคอมไพล์โปรแกรมภาษาซี
4. MPICH เวอร์ชัน 3.1.2 ที่เป็นมาตรฐาน MPI 3.0 ซึ่งใช้การสื่อสารระหว่างกลุ่มแบบไม่กีดขวางการรับส่งข้อมูล

วิธีการ

1. ส่วนต่อขยายของตัวชี้แนะคอมไพเลอร์ที่นำเสนอ

มาตรฐานโอเพนเอซีซี (OpenACC specification) มีแพร็กมาจำนวนมากสำหรับกำกับส่วนของโปรแกรมที่ทำงานแบบขนาน (Parallel Region) เช่น แพร็กมา parallel แพร็กมา loop และแพร็กมา kernel ซึ่งมีรายละเอียดการใช้งานที่ต่างกันออกไป แพร็กมา parallel ได้รับอิทธิพลมาจากแพร็กมา parallel ในโอเพนเอ็มพี และมีการทำงานคล้ายกันคือจะสร้างชุดของเทรดที่ทำงานแบบขนานขึ้นทันทีในส่วนของแพร็กมา parallel และเมื่อโค้ดด้านในมีแพร็กมา loop ก็จะจับคู่ระหว่างการทำงานของลูปและเทรด หรือสามารถใช้แพร็กมาแบบย่อ (Combined Construct) เป็นแพร็กมา parallel loop ก็จะให้การทำงานแบบเดียวกัน

แพร็กมา kernel จะได้รับอิทธิพลมาจากแพร็กมา region ของพีจีไอแอคเซราเรเตอร์ ซึ่งจะเป็นการแปลงส่วนของการทำงานนั้นให้เป็นเคอร์เนลที่ทำงานได้บนจีพียูโดยอัตโนมัติ แพร็กมา kernel ใช้งานได้สะดวกกว่า แต่จำเป็นต้องพึ่งพาความฉลาดในการวิเคราะห์ของคอมไพเลอร์ งานวิจัยนี้จะสนใจเฉพาะแพร็กมาแบบย่อ parallel loop ซึ่งทำงานเฉพาะกับการทำงานแบบขนานของลูปอย่างง่าย (Simple loop) ที่ซึ่งมีการกำหนดรอบการทำงานที่ชัดเจน และไม่มีการหยุดอย่างกะทันหัน



ภาพที่ 7 โมเดลการส่งข้อมูลระหว่างโฮสและดีไวซ์

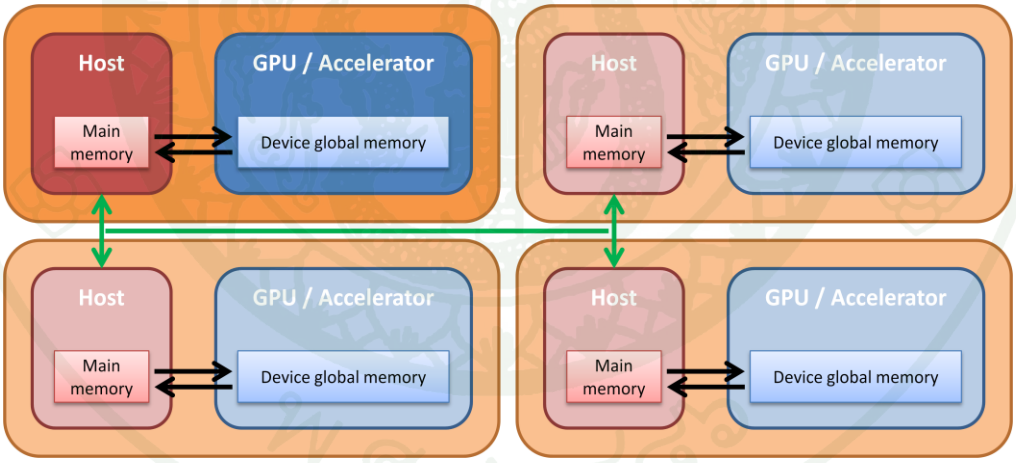
โมเดลของการส่งข้อมูลระหว่าง โฮสและจีพียูดีไวซ์ ในกรณีที่มีโฮสและดีไวซ์เพียงอย่างละตัว จะเป็นดัง ภาพที่ 7 คือมีการส่งข้อมูลไปกลับระหว่างหน่วยความจำของโฮสและดีไวซ์ ในมาตรฐานของโอเพนเอซีซีจะมีคลอสที่ใช้ควบคุมข้อมูลได้แก่ copyin, copyout และ copy (หมายถึงเป็นทั้ง copyin และ copyout) ซึ่งพารามิเตอร์ของคลอสจะเป็นชื่อตัวแปรที่จะทำการส่งข้อมูล

ระหว่างโฮสและดีไวซ์ หลังชื่อตัวแปรสามารถที่จะใส่สับอาร์เรย์ (Subarray) ซึ่งเป็นไวยากรณ์ที่ใช้กำหนดตำแหน่งและขนาดของอาร์เรย์ที่จะทำการส่ง สับอาร์เรย์มีรูปแบบการระบุคือ array-name[start:length] เมื่อ start คือออฟเซต (Offset) เริ่มต้นในการส่ง และ length คือขนาดที่จะส่ง

```
1  #pragma acc parallel loop copyin(A[0:n]) copyout(B[1:n-2])
2  for (i = 1; i < N-1; i++)
3     B[i] = ( A[i-1] + A[i] + A[i+1] ) / 3;
```

ภาพที่ 8 การใช้โอเพนเอซีซีบนโปรแกรม 3-Moving Average

บรรทัดที่ 1 ของภาพที่ 8 อาร์เรย์ A ที่อยู่ในคลอส copyin จะถูกส่งหน่วยความจำของโฮสไปยังจีพียู โดยเริ่มที่ตำแหน่ง 0 และส่งไปด้วยขนาด n (จำนวนเอเลเมนต์ของอาร์เรย์) ในขณะที่อาร์เรย์ B จะถูกส่งกลับมาจากจีพียู ซึ่งเริ่มต้นส่งกลับที่ตำแหน่ง 1 และมีจำนวนข้อมูลเป็น n-2



ภาพที่ 9 โมเดลการส่งข้อมูลของจีพียูคลัสเตอร์

ภาพที่ 9 แสดงถึงโมเดลหน่วยความจำของเครื่องจีพียูคลัสเตอร์ ในแต่ละโหนดจะมีหน่วยความจำของโฮสและดีไวซ์แยกกัน และแต่ละเครื่องจะส่งข้อมูลหากันผ่าน เน็ตเวิร์ก ตามโมเดลของระบบหน่วยความจำแบบกระจาย (Distributed-memory System) เพื่อให้สามารถระบุได้ว่าข้อมูลจะถูกกระจายไปในแต่ละโหนดได้อย่างไร ผู้วิจัยจึงได้นำเสนอส่วนขยายของสับอาร์เรย์เพื่อให้ระบุรูปแบบของการกระจายข้อมูลได้ โดยให้อยู่ในรูปแบบของ array [start:length

{dist_type}] เมื่อ dist_type คือรูปแบบการกระจายข้อมูล ซึ่งมีได้สองชนิดคือ broadcast และ partition การกระจายข้อมูลแบบ broadcast คือข้อมูลชุดนั้นจะถูกคัดลอกทั้งหมดและส่งไปโหนดที่เหลือ หรือก็คือการกระจายข้อมูลแบบ บรอดคาซท์ในเอ็มพีไอ แต่การกระจายข้อมูลแบบ partition คือการแบ่งข้อมูลออกเป็นส่วน และส่งแต่ละส่วนที่ไม่ซ้ำกันไปยังแต่ละโหนด หรือ การกระจายข้อมูลแบบสแคทเทอร์ มีข้อแม้คือผู้ใช้จะต้องกำหนดรูปแบบการกระจายข้อมูลได้เฉพาะมิติเดียวของอาร์เรย์เท่านั้น และในกรณีที่ไม่มีการระบุ รูปแบบโดยปริยายคือ broadcast ด้วยวิธีการที่นำเสนอนี้ ผู้พัฒนาโปรแกรมจะสามารถกำหนดการกระจายข้อมูลด้วยตัวเอง เพื่อให้เหมาะสมกับขั้นตอนวิธีมากที่สุด และใช้งานได้กับตัวแปรที่เป็นอาร์เรย์หลายมิติ

```
1  #pragma acc parallel loop copyin(A[0:n{partition}) copyout(B[1:n-2{partition}])
2  for (i = 1; i < N-1; i++)
3     B[i] = ( A[i-1] + A[i] + A[i+1] ) / 3;
```

ภาพที่ 10 การระบุรูปแบบการกระจายของส่วนต่อขยายบนโปรแกรม 3-Moving Average

จากโปรแกรมใน ภาพที่ 10 บรรทัดที่ 1 สับอาร์เรย์ของ A จะระบุชนิดการกระจายข้อมูลเป็น partition ซึ่งหมายความว่าตัวแปร A จะถูกกระจายแบบ สแคทเทอร์ ไปยังทุกเครื่อง และส่งข้อมูลขึ้นไปยังดีไวซ์ของแต่ละเครื่อง หลังจากประมวลผลเคอร์เนลเสร็จ ตัวแปร B แต่ละส่วนของแต่ละโหนดจะถูกรวมข้อมูลแบบแกทเธอร์กลับมาที่เครื่องหลัก

จีพียูใช้หน่วยความจำโกลบอลในการติดต่อสื่อสารกันระหว่างเทรด ทุกเทรดสามารถเข้าถึงและแก้ไขข้อมูลที่อยู่บนหน่วยความจำโกลบอล แต่เมื่อระบบถูกขยายเป็นระบบคลัสเตอร์ เทรดของจีพียูที่อยู่กันคนละโหนดจะไม่สามารถสื่อสารกันผ่านหน่วยความจำได้ ทำให้การนำไปใช้เพื่อแก้ปัญหาที่มีความขึ้นต่อกันของข้อมูลทำได้ยาก การสร้างคลอสสำหรับจัดการกับปัญหาเรื่องความขึ้นต่อกันของข้อมูล (Data Dependency) จึงจำเป็นสำหรับระบบจีพียูคลัสเตอร์ เนื่องจากในงานวิจัยนี้จะสนใจเฉพาะขั้นตอนวิธีที่มีรูปแบบการเข้าถึงข้อมูลไม่เปลี่ยนแปลงในขณะรัน ดังนั้นผู้วิจัยจึงเสนอคลอส in_pattern เพื่อใช้สำหรับระบุรูปแบบของการขึ้นต่อกันของข้อมูลของลูป for พารามิเตอร์ของคลอส in_pattern คือตัวแปร และรูปแบบของการขึ้นต่อกันของข้อมูล ซึ่งมีสองประเภทคือรูปแบบช่วง (Range Pattern) และรูปแบบระบุเอง (Specific Pattern) ไวยากรณ์ของรูปแบบช่วง คือ [lower_bound_index:upper_bound_index] และไวยากรณ์ของรูปแบบระบุเองคือ

[array_index_list] โดย array_index_list มีรูปแบบคือ $array\_index_1$,$array\_index_2$,…,$array\_index_n$ เมื่อ lower_bound_index upper_bound_index และ array_index เป็นค่าจำนวนเต็มที่หมายถึงอินเด็กซ์ของข้อมูลที่สัมพันธ์กับลูปในรอบนั้น

```
1  #pragma acc parallel loop copyin(A[0:n:partition]) copyout(B[1:n-2:partition]) \
2     in_pattern(A[-1:1])
3  for (i = 1; i < N-1; i++)
4     B[i] = ( A[i-1] + A[i] + A[i+1] ) / 3;
```

ภาพที่ 11 การใช้คลอส in_pattern ของส่วนต่อขยายบนโปรแกรม 3-Moving Average

ภาพที่ 11 แสดงตัวอย่างการใช้ส่วนต่อขยายของสับอาร์เรย์และคลอส in_pattern โปรแกรม 3-Moving Average อาร์เรย์ A ในรอบที่ i ต้องการข้อมูลของอาร์เรย์ตั้งตำแหน่งที่ i-1 จนถึง i+1 ดังนั้นในคลอส in_pattern จึงระบุด้วยรูปแบบช่วงเป็น [-1:1] รายละเอียดของแพร็กมาและคลอสทั้งหมดที่อิมพลีเมนต์ขึ้นเพื่อใช้ในงานวิจัยนี้อยู่ในตารางที่ 1 ตารางที่ 2 และตารางที่ 3

ตารางที่ 1 แพร็กมาของโอเพนเอซีซีที่อิมพลีเมนต์

โอเพนเอซีซีแพร็กมา	รายละเอียด
parallel loop	ระบุว่าคำสั่งลูป for ถัดไปสามารถทำงานแบบขนานได้
loop	ระบุว่าคำสั่งลูป for ถัดไปสามารถทำงานแบบขนานได้ โดยลูป for นี้จะต้องอยู่ในลูป for ที่กำกับไว้ด้วยแพร็กมา parallel loop
data	ระบุรูปแบบการทำงานกับข้อมูล

ตารางที่ 2 คลอสของแพร็กมา parallel loop ของโอเพนเอซีซีที่อิมพลีเมนต์

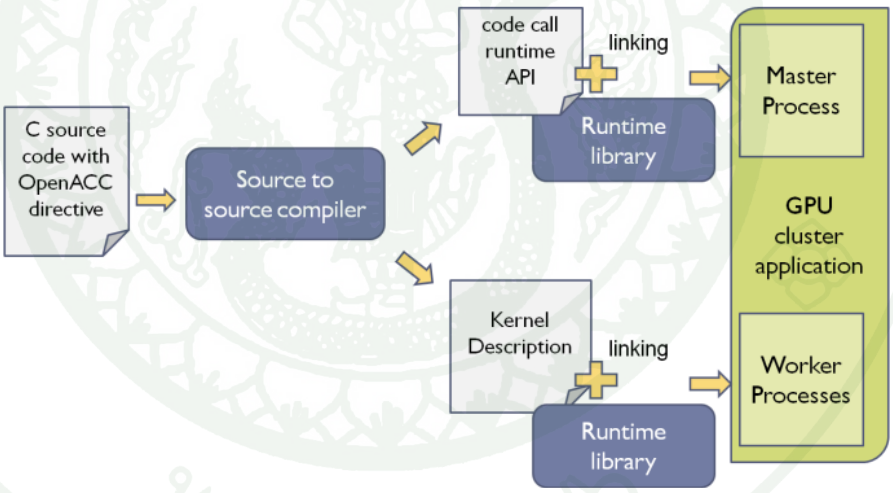
คลอสสำหรับ แพร็กมา parallel loop	รายละเอียด
copyin	ตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์จะต้องทำการส่งข้อมูลจากรูทโหนดไปยังหน่วยความจำบนดีไวซ์ของทุกโหนด
copyout	ตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์จะต้องทำการส่งข้อมูลหน่วยความจำบนดีไวซ์มายังโฮส และรวมข้อมูลมายังรูปโหนด เมื่อสิ้นสุดส่วนของการทำงาน
copy	ตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์จะถูกส่งข้อมูลเหมือนกับเป็นทั้ง copyin และ copyout
create	ตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์จะต้องถูกจองพื้นที่ไว้บนหน่วยความจำบนดีไวซ์ของทุกโหนด และจะไม่มีการย้ายข้อมูลใดๆเกิดขึ้นกับตัวแปรหรือสับอาร์เรย์นี้
present_or_copyin / pcopyin	ถ้าตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์ยังไม่ถูกส่งไปยังหน่วยความจำบนดีไวซ์ของแต่ละโหนด ให้ทำการส่งเหมือน copyin
present_or_copyout / pcopyout	ถ้าตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์ยังไม่ถูกส่งกลับจากหน่วยความจำบนดีไวซ์ของแต่ละโหนดไปยังรูทโหนด ให้ทำการส่งเหมือน copyout
present_or_copy / pcopy	ตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์จะเป็นทั้ง present_or_copyin / pcopyin และ present_or_copyout / pcopyout
present_or_create / pcreate	ถ้าตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์ยังไม่มีการจองพื้นที่บนหน่วยความจำบนดีไวซ์ของแต่โหนด ให้ทำการจองพื้นที่ให้ตัวแปรหรือสับอาร์เรย์นั้น
present	ตัวแปรหรือสับอาร์เรย์ที่อยู่ในลิสต์อยู่บนหน่วยความจำบนหน่วยความจำบนดีไวซ์ของแต่โหนดแล้ว ไม่ต้องมีการส่งข้อมูลใดๆอีก
private	กำหนดให้ตัวแปรนั้นต้องมีการประกาศใหม่บนเทรดของดีไวซ์
in_pattern (ส่วนต่อขยายที่นำเสนอเพิ่ม)	กำหนดรูปแบบการใช้งานข้อมูลของตัวแปรในลูปแต่ละรอบ
reduction	รีดิวซ์ตัวแปรสเกลาร์ที่อยู่ในคลอส reduction

ตารางที่ 3 คลอสของแพร็กมา data ของโอเพนเอซีซีที่อิมพลีเมนต์

คลอสสำหรับแพร็กมา data	รายละเอียด
copyin, copyout, copy, create, present_or_copyin / pcopyin, present_or_copyout / pcopyout, present_or_copy / pcopy, present_or_create / pcreate, present	เหมือนกับตารางที่ 2

2. ขั้นตอนการคอมไพล์และระบบรันไทม์

เพื่อให้สามารถใช้แพร็กมาและคลอสของโอเพนเอซีซีทั้งหมดที่สนับสนุน รวมไปถึงส่วนต่อขยายที่เพิ่มเติมขึ้นมา ผู้วิจัยจึงได้พัฒนาซอร์สทูซอร์สคอมไพเลอร์และรันไทม์ไลบรารี่ สำหรับแปลงโค้ดโอเพนเอซีซีให้ทำงานได้บนจีพียูคลัสเตอร์ ขั้นตอนเป็นไปตามภาพที่ 12



ภาพที่ 12 ขั้นตอนแปลงโค้ดที่นำเสนอสำหรับทำงานบนจีพียูคลัสเตอร์

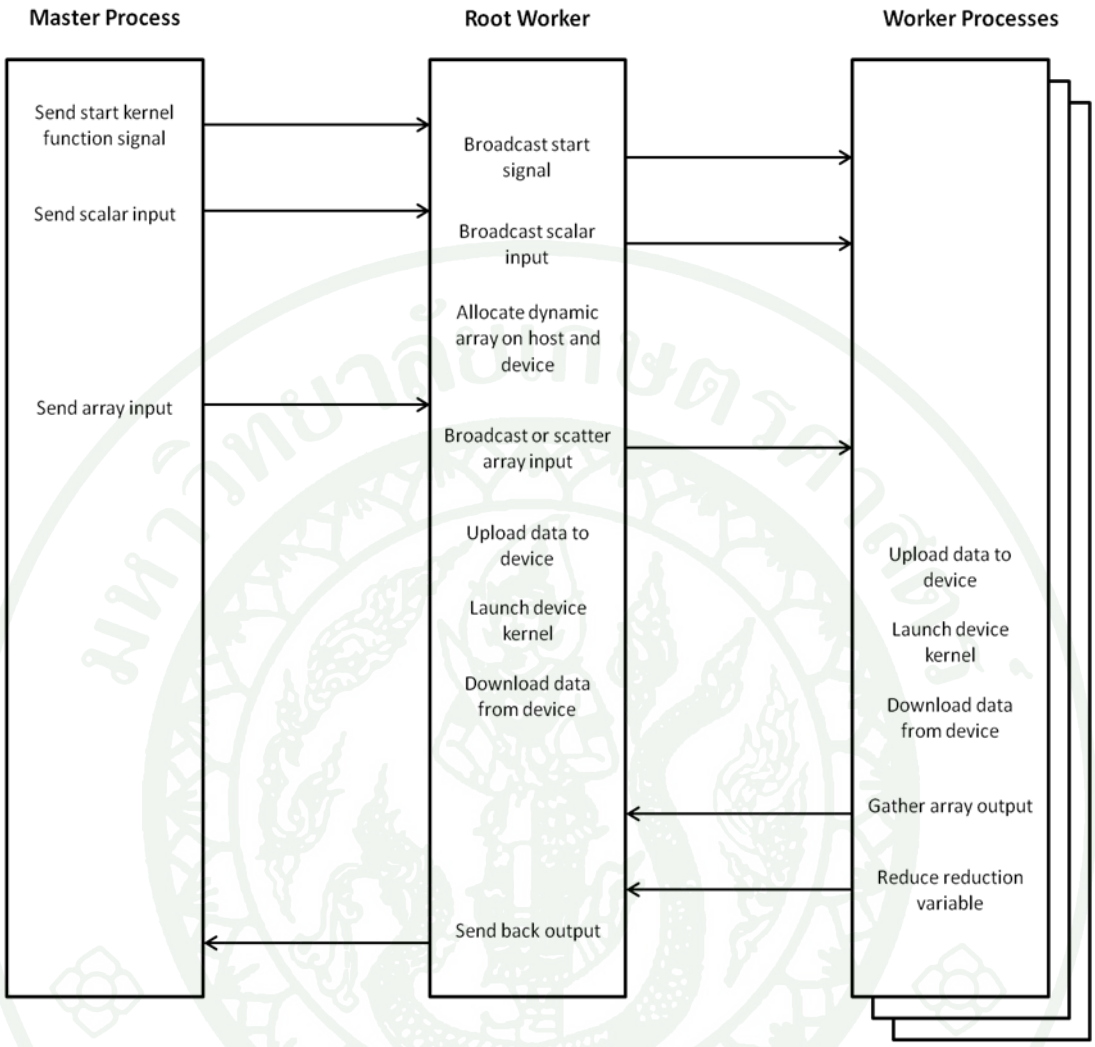
เริ่มต้นที่โค้ดภาษาซีที่มีตัวชี้แนะคอมไพเลอร์ของโอเพนเอซีซีจะถูกซอร์สทูซอร์สคอมไพเลอร์คอมไพล์ออกมาเป็นสองไฟล์ ได้แก่

1. ไฟล์โค้ดที่เก็บข้อมูลการทำงานของ เคอร์เนล และฟังก์ชันการทำงานของเวิร์กเกอร์ทั้งหมด เรียกไฟล์นี้ว่าไฟล์นิยามเคอร์เนล (Kernel Definition File) ซึ่งไฟล์นี้จะถูกคอมไพล์อีกครั้งเพื่อใช้เป็นเวิร์กเกอร์โพรเซส

2. ไฟล์สำหรับควบคุมการทำงาน ซึ่งจะถูกคอมไพล์อีกครั้งเป็นมาสเตอร์โพรเซส ที่จะคอยควบคุมขั้นตอนการทำงานทั้งหมดที่ไม่เกี่ยวกับการคำนวนแบบขนาน และเป็นตัวเรียกเวิร์กเกอร์โพรเซสทั้งหมดให้ทำงาน

การติดต่อสื่อสารระหว่างมาสเตอร์โพรเซสและเวิร์กเกอร์โพรเซสทั้งหมด (รวมถึงการส่งข้อมูล) จะถูกอิมพลีเมนต์ด้วย ซ็อกเก็ต (Socket) เพื่อให้มาสเตอร์โพรเซสและเวิรค์เกอร์โพรเซสสามารถทำงานร่วมกันได้แม้จะอยู่คนละเครื่อง โค้ดสำหรับทำงานบนจีพียูจะถูกสร้างขึ้นมาเป็นโค้ดเคอร์เนลของโอเพนซีแอล ซึ่งจะอยู่ในรูปของตัวแปรสตริงในไฟล์นิยามเคอร์เนล สำหรับแต่ละโปรแกรม ไฟล์นิยามเคอร์เนลสามารถมีได้หลายไฟล์ขึ้นกับจำนวนเคอร์เนลของโปรแกรมนั้น โดยที่สุดท้ายจะถูกนำมาคอมไพล์รวมกันให้เป็นโคดของเวิร์กเกอร์เพียงชุดเดียว ด้วยเครื่องมือที่สร้างขึ้นเรียกว่า worker-gen

ขั้นตอนการรันจะต้องทำการรันเวิร์กเกอร์โพรเซสทั้งหมดด้วยคำสั่ง mpirun ก่อน ซึ่งสามารถกำหนดจำนวนโพรเซส (หรือจำนวนโหนดของคลัสเตอร์) ได้ในคำสั่งนี้ เมื่อเวิร์กเกอร์โพรเซสทั้งหมดพร้อมแล้ว จึงเริ่มรันมาสเตอร์โพรเซสที่เป็นตัวควบคุมขั้นตอนการทำงานของโปรแกรมทั้งหมด ในเวิร์กเกอร์โพรเซสทั้งหมด จะมีเพียงหนึ่งโพรเซสเท่านั้นที่จะสามารถติดต่อกับมาสเตอร์โพรเซส และเป็นผู้กระจายข้อมูลจากมาสเตอร์โพรเซสไปยังเวิร์กเกอร์โพรเซสอื่น หรือรวมข้อมูลจากทุกเวิร์กเกอร์โพรเซสและส่งกลับไปยังมาสเตอร์โพรเซส จะเรียกโพรเซสนี้ว่า รูทเวิร์กเกอร์ (Root Worker) ภาพที่ 13 แสดงขั้นการทำงานร่วมกันของมาสเตอร์โพรเซสและเวิร์กเกอร์โพรเซสเพื่อประมวลผลแต่ละเคอร์เนล คำอธิบายโดยละเอียดจะอยู่ในหัวข้อของการแปลงแพร็กมา parallel for



ภาพที่ 13 ขั้นตอนการทำงานร่วมกันของมาสเตอร์และเวิร์กเกอร์โพรเซส

3. การใช้งานหน่วยความจำ

หน่วยความจำเป็นอีกส่วนสำคัญที่มีผลต่อประสิทธิภาพของโปรแกรมเชิงขนาน งานวิจัยนี้ใช้โมเดลหน่วยความจำแบบซ้ำ (Replicated Memory Model) ซึ่งหมายถึง สำหรับตัวแปรที่กำหนดรูปแบบการกระจายข้อมูลเป็นแบบ partition ทุกตัวจะถูกจองพื้นที่ไว้บนหน่วยความจำของทุกโหนด แต่ในการอัปเดตข้อมูลนั้นแต่ละโหนดจะจัดการในแต่ละส่วนที่ตนเองดูแลอยู่เท่านั้น การใช้งานหน่วยความจำแบบนี้มีข้อดีคือ ไม่จำเป็นต้องมีการแปลงโค้ดในส่วนอินเด็กซ์ของอาร์เรย์ ไม่จำเป็นต้องจองพื้นที่หน่วยความจำเพิ่มเติมสำหรับค่าขอบ (Boundary) ของข้อมูล และง่ายต่อการจัดการเมื่อมีการเปลี่ยนรูปแบบการกระจายข้อมูล โมเดลนี้ง่ายและช่วยลดเวลาในการประมวล

ผลรวม แต่ก็มีข้อเสียคือใช้พื้นที่หน่วยความจำจำนวนมาก แทนที่จะสามารถแบ่งภาระไปยังโหนดอื่นๆ และยังทำให้การสเกลเป็นไปได้ยากขึ้นเมื่อโปรแกรมมีการใช้อาร์เรย์ขนาดใหญ่ หรือใช้ข้อมูลจำนวนมาก

4. ขั้นตอนการแปลงและวิเคราะห์โค้ด

ผู้วิจัยได้ใช้โอเพนซอร์สที่มีชื่อว่าเมอร์คิวเรียม (Mercurium) ซึ่งเป็นซอร์สทูซอร์สคอมไพเลอร์ภาษา C/C++ สำหรับปรับปรุงเพื่อเพิ่มการสนับสนุนโอเพนเอซีซีและส่วนต่อขยายที่นำเสนอ ซอร์สโค้ดของเมอร์คิวเรียมง่ายต่อการปรับปรุง เนื่องจากมีการออกแบบเชิงวัตถุ มีการเตรียมฟังก์ชันที่จำเป็นสำหรับการแปลงโค้ดไว้ให้ใช้งานได้ทันที ขั้นตอนทั้งหมดของการแปลงโค้ดด้วยซอร์สทูซอร์สคอมไพเลอร์ที่นำเสนอขึ้น มีขั้นตอนดังนี้

1. เก็บข้อมูลคุณลักษณะของตัวแปร (Variable Properties) ของตัวแปรที่อยู่ในสโคปของแพร็กมา data
2. แปลงแพร็กมา data และคลอสที่เกี่ยวข้อง
3. เก็บข้อมูลลูป for ที่ใช้แพร็กมา parallel loop
4. เก็บข้อมูลคุณลักษณะของตัวแปร ของตัวแปรที่อยู่ในสโคปของแพร็กมา parallel loop
5. แปลงแพร็กมา parallel loop และคลอสที่เกี่ยวข้อง
6. สร้างไฟล์นิยามเคอร์เนลและไฟล์มาสเตอร์โพรเซส

4.1 คุณลักษณะของตัวแปร ขั้นตอนแรกของการแปลงโค้ดคือการเก็บข้อมูลคุณลักษณะของตัวแปรทั้งหมดที่อยู่ส่วนของโค้ดที่จะถูกแปลงเป็นเคอร์เนลก่อน คุณสมบัติที่ต้องการใช้ได้แก่

4.1.1 Is_input – ตัวแปรเป็นข้อมูลเข้าของเคอร์เนลหรือไม่ จะมีค่าเป็นจริงเมื่อตัวแปรถูกระบุใน คลอส copy copyin pcopy หรือ pcopyin

4.1.2 Is_output – ตัวแปรเป็นข้อมูลส่งออกของเคอร์เนลหรือไม่ จะมีค่าเป็นจริงเมื่อตัวแปรถูกระบุในคลอส copy copyout pcopy หรือ pcopyout

4.1.3 Is_present – ตัวแปรมีค่าอยู่ในดีไวซ์แล้วและไม่ต้องทำการส่งข้อมูลหรือไม่ จะมีค่าเป็นจริงเมื่อตัวแปรถูกระบุในคลอส present pcopy pcopyin หรือ pcopyout

4.1.4 Is_private – ตัวแปรเป็นตัวแปรแบบ private ในการทำงานของเคอร์เนลหรือไม่ จะมีค่าเป็นจริงเมื่อตัวแปรถูกระบุในคลอส private

4.1.5 Is_def – ตัวแปรถูกเขียนข้อมูลในการทำงานของเคอร์เนลหรือไม่ จะมีค่าเป็นจริงเมื่อปรากฏอยู่ที่ด้านซ้าย (LHS) ใน Assignment Expression ที่อยู่ในการทำงานของเคอร์เนล

4.1.6 Is_use – ตัวแปรถูกอ่านข้อมูลในการทำงานของเคอร์เนลหรือไม่ จะมีค่าเป็นจริงเมื่อปรากฏอยู่ในที่ใดๆที่ไม่ใช่ด้านซ้ายใน Assignment Expression ที่อยู่ในการทำงานของเคอร์เนล

4.1.7 Is_reduction – ตัวแปรเป็นชนิดรีดักชันหรือไม่ จะมีค่าเป็นจริงเมื่อตัวแปรถูกระบุใน คลอส reduction

4.1.8 Reduction_type – ชนิดของการรีดักชันหรือโอเปอเรเตอร์ที่ใช้ในการรีดิวซ์ ได้จากโอเปอเรเตอร์ในคลอส reduction

4.1.9 Array_size – ขนาดของตัวแปร (กรณีที่เป็นอาร์เรย์) ได้จากสับอาร์เรย์ในคลอสตระกูล copy

4.1.10 Paritition_dimension – มิติของอาร์เรย์ที่มีการกระจายข้อมูล ได้จากตำแหน่งของสับอาร์เรย์ในคลอสตระกูล copy ที่มีรูปแบบการกระจายปรากฏอยู่

4.1.11 Access_pattern – รูปแบบการขึ้นต่อกันของข้อมูลของตัวแปรในแต่ละรอบของลูป ได้จากคลอส in_pattern

4.2 การแปลงแพร็กมา parallel loop แพร็กมา parallel loop จะต้องใช้กับลูป for ที่สามารถเปลี่ยนเป็นการประมวลผลในรูปแบบ SIMD ได้ (รายละเอียดรูปแบบของลูปชนิดนี้มีระบุไว้ในมาตรฐานโอเพนเอ็มพี)คอมไพเลอร์จะแปลงลูปให้กลายเป็นการทำงานแบบขนานเชิงลำดับ (Hierarchical Parallelism) โดยที่เลเวลแรกของการทำงานแบบขนาน จะเป็นการแบ่งรอบการทำงานของลูปไปให้แต่ละคลัสเตอร์โหนด โดยพยายามทำให้เท่ากันมากที่สุด ด้วยการดูจากข้อมูลรอบการทำงานของลูป และแบ่งส่วนของหน่วยความจำที่เก็บตัวแปรไปยังแต่ละคลัสเตอร์โหนดด้วยรูปแบบการกระจายข้อมูลที่ผู้ใช้กำหนดไว้ในสับอาร์เรย์

ตัวอย่างเช่น ถ้าโปรแกรมทำงานอยู่บนคลัสเตอร์ที่มีสามโหนด และลักษณะของลูป for คือ for (i=0 ; i<32; i++) โหนดแรกจะถูกแบ่งให้ทำงานในรอบที่ [0-10] ของลูป โหนดถัดๆไปจะได้การทำงานรอบที่ [11-21] และ [22-31] ตามลำดับ ถ้าตัวแปร a มีการกำหนดสับอาร์เรย์ไว้ใน

รูปแบบ a[0:32{partition}][0:5] แต่ละโหนดก็จะได้รับข้อมูลในช่วง a[0:10][0:5], a[11:21][0:5] และ a[22:31][0:5] ตามลำดับ

การแบ่งลูปไปยังจีพียูเทรดเป็นเลเวลถัดไปของการทำงานแบบขนาน หลังจากที่เลเวลแรกกระจายรอบการทำงานของลูปมายังแต่ละคลัสเตอร์โหนด รอบการทำงานที่ถูกกระจายมาจะถูกแปลงเป็นเคอร์เนลของโอเพนซีแอลเพื่อรันบนจีพียู วิธีการแบ่งรอบการทำงานของลูปไปยังเทรดจะใช้วิธีการในงานวิจัยก่อนหน้าของผู้วิจัย (Makpaisit and Marurnsith, 2011)

```
1. Declare variables including the generated variables
2. Broadcast the scalar variables to all workers
3. Allocate the array variables
4. Initialize the generated variables
5. Allocate the reduction variables
6. Scatter the array variables to all workers
7. Launch the kernel
8. Gather the array variables from all workers
9. Reduce the reduction variables from all workers
```

ภาพที่ 14 เทมเพลตสำหรับการแปลงแพร็กมา parallel loop ให้เป็นเวิร์กเกอร์ฟังก์ชัน

เวิร์กเกอร์ฟังก์ชันบนไฟล์นิยามเคอร์เนลจะถูกสร้างตามเทมเพลตใน ภาพที่ 14 แต่ละแพร็กมา parallel loop จะถูกสร้างเวิร์กเกอร์ฟังก์ชันหนึ่งตัว แต่ละขั้นตอนในเทมเพลตสามารถอธิบายได้ดังนี้

1. นิยามตัวแปรที่ใช้ในเคอร์เนล

2. กระจายข้อมูลแบบบรอดคาซท์ตัวแปรสเกลาร์ทั้งหมดที่ใช้ในเคอร์เนล เพื่อให้ทุกเวิร์กเกอร์โพรเซสได้ค่าของตัวแปรที่จำเป็น ในขั้นตอนนี้ใช้รันไทม์ไลบรารีฟังก์ชันที่สร้างขึ้นที่ชื่อ _EnqueueBcastScalar ในการส่งตัวแปรสเกลาทั้งหมด

3. พื้นที่หน่วยความจำของตัวแปรอาร์เรย์จะต้องถูกจองพื้นที่ (Memory Allocation) ก่อนที่จะมีการใช้งาน ฟังก์ชัน _MallocND จะจองพื้นที่ทั้งบนโฮสและดีไวซ์ให้กับตัวแปรอาร์เรย์ และสร้างแมปสำหรับเก็บข้อมูลว่าตัวแปรแต่ละตัวถูกจองพื้นที่ไว้แล้วหรือไม่ และใช้ตำแหน่งในหน่วยความจำบนมาสเตอร์โพรเซสของตัวแปรนั้นเป็นคีย์

4. กำหนดค่าเริ่มต้นให้กับตัวแปรใหม่ที่คอมไพเลอร์สร้างขึ้น ตัวแปรที่ถูกสร้างขึ้นจากขั้นตอนการแปลงโค้ด เช่น อินเด็กซ์เริ่มต้นและสิ้นสุดของลูปบนแต่ละคลัสเตอร์โหนด ขนาด และจำนวนของเคอร์เนลเวิร์กกรุป เป็นต้น

5. จองพื้นที่ให้กับตัวแปรรีดักชัน สำหรับเก็บค่ารีดิวซ์ย่อยในแต่ละโหนด โดยผ่านการเรียกใช้ฟังก์ชัน _MallocReduceScalar ที่สร้างขึ้น

6. กระจายข้อมูลจากรูทเวิร์กเกอร์ไปยังเวิร์กเกอร์โพรเซสอื่นโดยเรียกใช้ฟังก์ชัน _EnqueueBcastND สำหรับการกระจายข้อมูลอาร์เรย์แบบบรอดคาซท์ และเรียกใช้ _EnqueueScatterND สำหรับกระจายข้อมูลอาร์เรย์แบบสแคทเทอร์ โดยจะต้องส่งมิติที่แบ่งไปให้ฟังก์ชัน _EnqueueScatterND ด้วย

7. เมื่อเวิร์กเกอร์โพรเซสได้ข้อมูลที่ต้องการครบแล้ว จะส่งพารามิเตอร์ที่เคอร์เนลต้องการไปให้ และเรียกให้เคอร์เนลทำงาน

8. รวบรวมข้อมูลอาร์เรย์ที่ประมวลผลเสร็จแล้วจากทุกเวิร์กเกอร์โพรเซสด้วยฟังก์ชัน _EnqueueGatherND

9. ข้อมูลส่งออกอีกประเภทที่ไม่ใช่การรวมข้อมูลแบบแกทเธอร์คือตัวแปรรีดักชัน ซึ่งขั้นตอนสุดท้ายจะต้องรีดิวซ์กลับมายังรูทเวิร์กเกอร์ ฟังก์ชันที่รับผิดชอบคือ _EnqueueReduceScalar

บนไฟล์มาสเตอร์โพรเซส แพร็กมา parallel loop และลูป for ที่เกี่ยวข้องจะถูกแปลงเป็นการส่งซิกแนลไปยังรูทเวิร์กเกอร์ เพื่อเรียกให้เริ่มทำงานเวิร์กเกอร์ฟังก์ชัน จากนั้นจะทำการส่งข้อมูลของตัวแปรสเกลาร์และอาร์เรย์ไปยังรูทเวิร์กเกอร์ โดยที่ลำดับการส่งข้อมูลจากรูทเวิร์กเกอร์ไปยังเวิร์กเกอร์ทั้งหมดจะต้องตรงกัน จากนั้นมาสเตอร์โพรเซสจะทำการเรียกฟังก์ชันสำหรับรับข้อมูลส่งออกจากเวิร์กเกอร์ (รวมไปถึงตัวแปรรีดักชัน) เทมเพลตสำหรับการแปลงแพร็กมา parallel loop บนไฟล์มาสเตอร์เป็นไปตามภาพที่ 15

```
1. Send the call kernel function signal to worker
2. Send the scalar input variables to root worker
3. Send the array input variables to root worker
4. Receive the output variables from root worker
```

ภาพที่ 15 เทมเพลตสำหรับการแปลงแพร็กมา parallel loop บนไฟล์มาสเตอร์โพรเซส

เมื่อคอมไพเลอร์พบลูป for ของแพร็กมา loop ที่อยู่ในสโคปของแพร็กมา parallel loop จะทำการรวมเข้ากับลูป for ของแพร็กมา parallel loop ภายนอกให้อัตโนมัติ แต่การแบ่งงานของแต่ละโหนดจะยังคงอ้างอิงถึงลูปภายนอกเหมือนเดิม

4.3 การแปลงแพร็กมา data รันไทม์ไลบรารี่ที่สร้างขึ้นจะต้องมีฟังก์ชันสำหรับทดสอบว่าตัวแปรมีข้อมูลล่าสุดแล้วหรือไม่ ผู้วิจัยได้สร้างแมปสำหรับเก็บข้อมูลตัวแปรทั้งหมด และฟังก์ชันสำหรับตรวจสอบว่าตัวแปรใดที่ไม่จำเป็นต้องมีการส่งอีก เพื่อเป็นการสนับสนุนคลอสในตระกูล present

คอมไพเลอร์จะสร้างฟังก์ชันสำหรับแพร็กมา data บนไฟล์นิยามเคอร์เนลสองฟังก์ชัน

1. ฟังก์ชัน send สำหรับรับอาร์เรย์ที่เป็นข้อมูลนำเข้าจากมาสเตอร์โพสเซส โดยที่ตัวแปรอาร์เรย์ทุกตัวที่ปรากฏอยู่ในแพร็กมา data จะต้องเรียกฟังก์ชัน _LockTransfer จากรันไทม์ไลบรารี่ที่สร้างขึ้น หลังจากมีการส่งข้อมูลเสร็จแล้ว

2. ฟังก์ชัน receive สำหรับส่งข้อมูลในอาร์เรย์ที่รวมแบบแกทเธอร์ กลับไปยังมาสเตอร์โพรเซส หลังจากการส่งข้อมูลกลับแล้วจะเรียกฟังก์ชัน _UnlockTransfer เพื่อให้สามารถส่งข้อมูลใหม่ได้อีกครั้ง

บนไฟล์มาสเตอร์โพรเซสคอมไพเลอร์จะแทรกฟังก์ชันสำหรับส่งซิกแนลไปยังรูทเวิร์กเกอร์ เพื่อเรียกฟังก์ชัน send บนเวิร์กเกอร์ และส่งซิกแนลเรียกฟังก์ชัน receive หลังจบแพร็กมา data การแปลงโค้ดของแพร็กมา data จะแปลงแบบรีเคอร์ซีฟ (Recursive) ด้วยการแปลงและสร้างโค้ดใหม่ให้กับแพร็กมาและชุดคำสั่งด้านนอกสุดก่อน แล้วจึงแปลงแพร็กมาที่อยู่ด้านในต่อไป

4.4 การแปลงคลอสข้อมูล ข้อมูลตัวแปรอาร์เรย์ที่อยู่ในคลอส copyin และ copy จะถูกกระจายแบบ บรอดคาซท์ หรือ สแคทเทอร์ (ตามที่ผู้ใช้ระบุ) ไปยังหน่วยความจำของทุกเวิร์กเกอร์โพรเซส จากนั้นทำการส่งข้อมูลไปยังหน่วยความจำบนจีพียู ซึ่งการทำงานเหล่านี้จะถูกอิมพลีเมนต์อยู่ใน _EnqueueBcastND และ _EnqueueScatterND ของไลบรารี่ที่สร้างขึ้น สำหรับตัวแปรในคลอส copyout และ copy จะทำการส่งจากจีพียูลงมายังเครื่องเครื่องโฮสเมื่อสิ้นสุดการทำงานของเคอร์เนล และตามด้วยการ การรวมข้อมูลแบบแกทเธอร์ ไปยังรูทเวิร์กเกอร์ โดยอาศัยฟังก์ชัน _EnqueueGatherND

สำหรับตัวแปรที่อยู่ในคลอส present_or_* ของแพร็กมา data ฟังก์ชัน _EnqueueBcastND, _EnqueueScatterND และ _EnqueueGatherND จะทำงานเหมือนคลอส copy* ถ้าตัวแปรไม่ถูกล็อคไม่ให้ส่งข้อมูล แต่ถ้าหากตัวแปรถูกล็อคนั่นหมายถึงตัวแปรนั้นไม่จำเป็นต้องมี

การส่งข้อมูลใดๆอีก ยกเว้น boundary ที่ระบุไว้ในคลอส in_pattern ซึ่งจะต้องมีการส่งข้อมูลเสมอ ทั้งนี้เนื่องจากคลอส present เพียงใช้บอกว่าข้อมูลของสับอาร์เรย์ที่ถูกแบ่งนั้นมีอยู่แล้วในแต่ละโหนด แต่ไม่ได้รวมถึง ค่าขอบซึ่งอยู่และถูกคำนวนค่าบนโหนดอื่น

4.5 การแปลงคลอส reduction คอมไพเลอร์จะสร้างตัวแปรโลคอลบนโฮสสำหรับแต่ละเวิร์กเกอร์โพรเซส เพื่อเก็บค่าที่รีดิวซ์บนจีพียู และส่งกลับไปยังรูทเวิร์กเกอร์และมาสเตอร์โพรเซสด้วยฟังก์ชัน MPI_Reduce ของเอ็มพีไอ ในส่วนของการรีดิวซ์บนจีพียูจะใช้เทคนิคการรีดิวซ์แบบขนาน (Parallel Reduction)

5. รันไทม์ไลบรารี่ฟังก์ชัน

รันไทม์ไลบรารี่ฟังก์ชันทั้งหมดที่พัฒนาขึ้นจะแสดงรายละเอียดไว้ใน ตารางที่ 4

ตารางที่ 4 รันไทม์ไลบรารี่ฟังก์ชันในระบบ

ฟังก์ชัน	รายละเอียด
_MallocND	ทำการจองพื้นที่ให้กับอาร์เรย์ทั้งบนโฮสและดีไวซ์ของทุกโหนดถ้าหากอาร์เรย์นั้นยังไม่ได้ถูกจองพื้นที่ และส่งตำแหน่งของอาร์เรย์กลับออกมา
_MallocReduceScalar	จองพื้นที่บนหน่วยความจำของดีไวซ์สำหรับตัวแปรรีดักชันบนดีไวซ์
_FreeReduceScalar	ปล่อยพื้นที่จองการจองสำหรับตัวแปรรีดิวซ์
_EnqueueBcastScalar	การจายข้อมูลของตัวแปรสเกลาร์แบบบรอดคาซท์ไปยังทุกเวิร์กเกอร์โพรเซส
_FinishBcastScalar	ฟังก์ชัน _EnqueueBcastScalar ที่เรียกก่อนหน้านี้ทั้งหมดจะต้องทำงานจนเสร็จสมบูรณ์
_EnqueueReduceScalar	รีดิวซ์ค่าของตัวแปรรีดักชัน ด้วยการส่งค่ารีดิวซ์กลับจากดีไวซ์และรีดิวซ์อีกครั้งในระดับคลัสเตอร์
_FinishReduceScalar	ฟังก์ชัน _EnqueueReduceScalar ที่เรียกก่อนหน้านี้ทั้งหมดจะต้องทำงานจนเสร็จสมบูรณ์
_EnqueueBcastND	กระจายข้อมูลของตัวแปรอาร์เรย์แบบบรอดคาซท์ไปยังหน่วยความจำบนจีพียูของทุกเวิร์กเกอร์โพรเซส ถ้าตัวแปรนั้นไม่ได้ถูกล็อค
_EnqueueScatterND	กระจายข้อมูลของตัวแปรอาร์เรย์แบบสแคทเทอร์ไปยังหน่วยความจำบนจีพียูของทุกเวิร์กเกอร์โพรเซส ถ้าตัวแปรนั้นไม่ได้ถูกล็อค และแลกเปลี่ยนข้อมูล boundary ไม่ว่าตัวแปรนั้นจะถูกล็อคหรือไม่ก็ตาม
_FinishDistributeArray	ฟังก์ชัน _EnqueueBcastND และ _EnqueueScatterND ที่เรียกก่อนหน้านี้ทั้งหมดจะต้องทำงานจนเสร็จสมบูรณ์
_EnqueueGatherND	การรวมข้อมูลแบบแกทเธอร์กลับจากหน่วยความจำบนจีพียูของทุกเวิร์กเกอร์โพรเซส ไปยังรูทเวิร์กเกอร์ ถ้าตัวแปรนั้นไม่ได้ถูกล็อค
_FinishGatherArray	ฟังก์ชัน _EnqueueGatherND ที่เรียกก่อนหน้านี้ทั้งหมดจะต้องทำงานจนเสร็จสมบูรณ์
_LockTransfer	ล็อคไม่ให้ตัวแปรอาร์เรย์ถูกส่งข้อมูล ยกเว้นค่า boundary สำหรับถูกแลกเปลี่ยนที่ระบุในคลอส in_pattern
_UnlockTransfer	ปลดล็อคเพื่อให้ตัวแปรอาร์เรย์สามารถถูกส่งข้อมูลได้อีกครั้ง

6. ตัวอย่างการแปลงโค้ด

ส่วนนี้จะแสดงขั้นตอนการแปลงโค้ดของซอร์สทูซอร์สคอมไพเลอร์ที่สร้างขึ้น โดยมีตัวอย่างเป็นโปรแกรมคูณเมทริกซ์เริ่มต้นด้วยการนำโค้ดเชิงลำดับมาเพิ่มตัวชี้แนะคอมไพเลอร์โอเพนเอซีซีและส่วนต่อขยายที่เสนอดังภาพที่ 16 ไฟล์มาสเตอร์โพรเซสจะแทนที่แพรกมา parallel loop และลูป for ด้วยคำสั่งเรียกรันไทม์ไลบรารี่ดังภาพที่ 17

```
#pragma acc parallel loop private(k) \
    copyin(A[0:N{partition}][0:N]) copyin(B[0:N][0:N]) \
    copyout(C[0:N{partition}][0:N])
    for (i = 0; i < N; i++)
#pragma acc loop
        for (j = 0; j < N; j++) {
            C[i][j] = 0.0f;
            for (k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j]; }
```

ภาพที่ 16 ตัวอย่างโปรแกรมคูณเมทริกซ์โดยใช้โอเพนเอซีซี

```
_SendCallFuncMsg(698324);
_SendInputMsg((void *)&N, sizeof(int));
_SendConstInputMsg((long long)&(A[0][0]));
_SendConstInputMsg((long long)&(B[0][0]));
_SendConstInputMsg((long long)&(C[0][0]));
_SendInputNDMsg(&(A[0][0]),_DOUBLE_T,0,N-1,1,0,0,2,0,N,N);
_SendInputMsg((void *)B[0],sizeof(double) * N * N);
_RecvOutputNDMsg(&(C[0][0]),_DOUBLE_T,0,N-1,1,0,0,2,0,N,N);
```

ภาพที่ 17 ตัวอย่างโค้ดโปรแกรมคูณเมทริกซ์ที่ถูกแปลงบนไฟล์มาสเตอร์โพรเซส

```
kernel void kernel_698324(int N,__global const double *A,
    global const double *B,__global double *C,
    int _local_i_start, int _local_i_end, int _loop_step,
    int _j_start, int _j_end, int _inner_step) {
int j, k, outer_size = (local_i_end-local_i_start+1)/_loop_step;
int inner_size = (j_end-j_start+1) / _inner_step;
int loop_size = outer_size * inner_size;
int i = ((get_global_id(0) / inner_size) * _loop_step) +
_local_i_start;
j = ((get_global_id(0)%inner_size)*inner_step)+_j_start;
if (i <= N - 1) {
    C[i * N + j] = 0.0f;
    for (k = 0; k < N; k++)
        C[i * N + j]+= A[i * N + k] * B[k * N + j]; }
}
```

ภาพที่ 18 ตัวอย่างโอเพนเอซีซีเคอร์เนลที่ถูกสร้างขึ้นบนไฟล์นิยามเคอร์เนล

ไฟล์นิยามเคอร์เนลที่ถูกสร้างขึ้นสำหรับ เวิร์กเกอร์ โพรเซสจะมีนิยามของโอเพนซีแอลเคอร์เนลเพื่อทำงานบนจีพียู และมีเวิร์กเกอร์ฟังก์ชันที่เป็นส่วนควบคุมขั้นการทำงานและการทำงานร่วมกันของเวิร์กเกอร์โพรเซส แสดงตัวอย่างไว้ในภาพที่ 18 และภาพที่ 19 ตามลำดับ

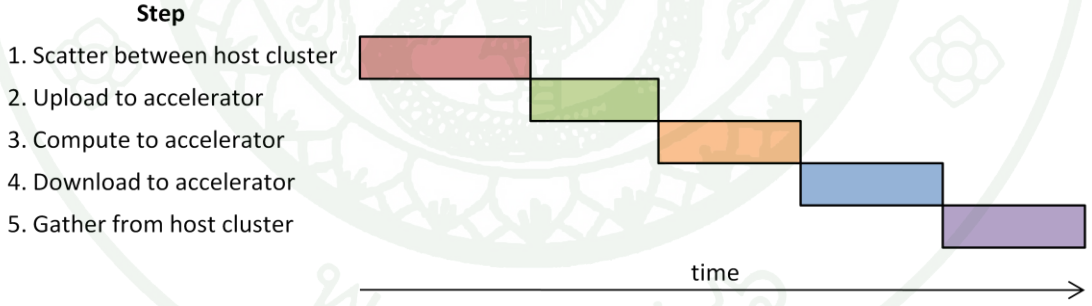
```
void _Function_698324() {
…
/* Broadcast Scalar Value */
_EnqueueBcastScalar(&N, _INT_T);
_EnqueueBcastScalar(&_id_A, _LL_INT_T);
_EnqueueBcastScalar(&_id_B, _LL_INT_T);
_EnqueueBcastScalar(&_id_C, _LL_INT_T);
_FinishBcastScalar();
/* Allocate Array Memory */
_Malloc2D(&A,&_cl_mem_A,_id_A,_DOUBLE_T,N,N,_MEM_READ_ONLY);
_Malloc2D(&B,&_cl_mem_B,_id_B,_DOUBLE_T,N,N,_MEM_READ_ONLY);
_Malloc2D(&C,&_cl_mem_C,_id_C,_DOUBLE_T,N,N,_MEM_READ_WRITE);
/* Initialize Generated Variables */
_local_i_start = _CalcLocalLoopStart(0, N - 1, 1);
_local_i_end = _CalcLocalLoopEnd(0, N - 1, 1);
_loop_step = 1;
_j_start = 0; _j_end = N - 1; _inner_step = 1;
_work_item_num = _CalcSubSize(_gen_loop_size,__num_proc, __rank, 1);
_work_group_item_num = 64;
_global_item_num = _CalcGlobalItemNum(_work_item_num,
_work_group_item_num);
/* Distribute Array Memory */
_EnqueueScatterND((void *)A[0],_cl_mem_A,_id_A,_DOUBLE_T,
_MEM_READ_ONLY,0,N-1,1,0,0,1,1,2,0,N,N);
_EnqueueBcastND((void *)B[0],_cl_mem_B,_id_B,_DOUBLE_T,1, 1,2,N,N);
_FinishDistributeArray();
/* Compute Workload */
_kernel = _CreateKernel("_kernel_698324");
_SetKernelArg(_kernel,0,sizeof(int),(void *)&N);
_SetKernelArg(_kernel,1,sizeof(cl_mem),(void *)&_cl_mem_A);
_SetKernelArg(_kernel,2,sizeof(cl_mem),(void *)&_cl_mem_B);
_SetKernelArg(_kernel,3,sizeof(cl_mem),(void *)&_cl_mem_C);
_SetKernelArg(_kernel,4,sizeof(int),(void *)&_local_i_start);
_SetKernelArg(_kernel,5,sizeof(int),(void *)&_local_i_end);
_SetKernelArg(_kernel,6,sizeof(int),(void *)&_loop_step);
_SetKernelArg(_kernel,7,sizeof(int),(void *)&_j_start);
_SetKernelArg(_kernel,8,sizeof(int),(void *)&_j_end);
_SetKernelArg(_kernel,9,sizeof(int),(void *)&_inner_step);
_LaunchKernel(_kernel,&_global_item_num, &_work_group_item_num);
_ClearKernel(_kernel);
/* Gather Array Memory */
_EnqueueGatherND((void *)C[0],_cl_mem_C,_id_C,_DOUBLE_T,
_MEM_READ_WRITE,0,N-1,1,0,0,1,1,2,0,N,N);
_FinishGatherArray();
}
```

ภาพที่ 19 ตัวอย่างเวิร์กเกอร์ฟังก์ชันที่สร้างขึ้นบนไฟล์นิยามเคอร์เนล

7. การปรับปรุงสมรรถนะ

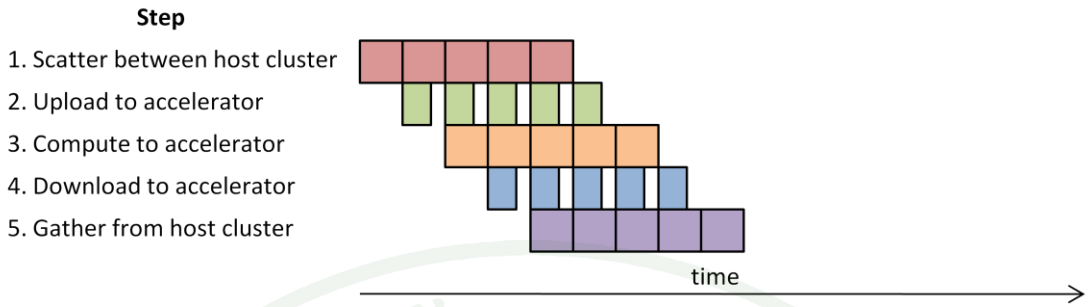
7.1 การแบ่งเคอร์เนล (Kernel Spliting) งานวิจัยนี้จะใช้เทคนิคการแบ่งเคอร์เนลสำหรับการปรับปรุงสมรรถนะอัตโนมัติในคอมไพเลอร์ เนื่องจากเป็นวิธีที่มีประสิทธิภาพ สามารถลดเวลาการประมวลผลลง ด้วยการทำให้เกิดการทำงานพร้อมกันของการส่งข้อมูลและการประมวลผล อีกทั้งยังสามารถทำเป็นระบบอัตโนมัติได้ง่าย เนื่องจากเป็นการแบ่งการทำงานของข้อมูลทีละส่วน ซึ่งสามารถใช้ข้อมูลจากตัวชี้แนะคอมไพเลอร์ที่กำกับไว้ ทำให้คอมไพเลอร์รู้ความขึ้นต่อกันของข้อมูลอยู่ก่อนแล้ว และสามารถแบ่งการทำงานออกเป็นส่วนได้โดยง่าย

ขั้นตอนและเวลาการทำงานของแต่ละเคอร์เนลบนระบบจีพียูคลัสเตอร์เป็นไปดัง ภาพที่ 20 เริ่มต้นรูทเวิร์กเกอร์การกระจายข้อมูลไปให้ทุกเวิร์กเกอร์โพรเซสก่อน หลังจากที่แต่ละโหนดได้ข้อมูลในส่วนที่ตนเองต้องการแล้ว จึงจะสามารถส่งไปยังหน่วยความจำของจีพียูได้ และจึงสามารถเริ่มทำการประมวลผล ส่งข้อมูลกลับมายังหน่วยความจำของเครื่องโฮส จากนั้นจึงรวบรวมข้อมูลระหว่างโหนดกลับไป ขั้นตอนทั้งหมดมีการขึ้นต่อกันของการทำงาน ซึ่งต้องรอให้ขั้นตอนก่อนหน้าทำงานเสร็จก่อนจึงจะเริ่มขึ้นตอนถัดไปได้



ภาพที่ 20 ขั้นตอนและเวลาการทำงานของโปรแกรมบนจีพียูคลัสเตอร์

แต่ด้วยเทคนิคการแบ่งเคอร์เนล ข้อมูลและรอบการทำงานของลูปจะถูกแบ่งออกเป็นส่วนย่อย ข้อมูลส่วนแรกจะถูกกระจายไปยังแต่ละโหนดก่อน จากนั้นระหว่างที่กำลังส่งข้อมูลส่วนแรกไปยังหน่วยความจำของจีพียู ข้อมูลส่วนที่สองก็จะถูกกระจายไปยังทุกโหนด ขั้นตอนจะเป็นแบบนี้ไปเรื่อยๆ ทำให้เกิดการทำงานพร้อมกันของการส่งข้อมูลและการประมวลผลดังภาพที่ 21



ภาพที่ 21 ขั้นตอนและเวลาการทำงานของโปรแกรมเมื่อใช้เทคนิคการแบ่งเคอร์เนล

7.2 การอิมพลีเมนต์ เพื่อที่จะแบ่งการทำงานของเคอร์เนลและการส่งข้อมูลออกเป็นส่วนย่อย และสามารถเก็บสถานะการทำงานได้ ผู้วิจัยได้สร้างโครงสร้างข้อมูลสำหรับเก็บข้อมูลของแต่ละเคอร์เนลในขณะทำงาน โดยข้อมูลที่มีการบันทึกไว้ได้แก่ หมายเลขเคอร์เนล ข้อมูลตัวแปรลูป ขนาดของข้อมูลทุกตัวที่ต้องส่ง ขนาดของข้อมูลที่ส่งขึ้นไปแล้ว จำนวนรอบของลูปที่ทำงานไปแล้ว โดยได้สร้างแมปของแต่ละเคอร์เนลไว้ ทุกครั้งที่มีการเริ่มต้นเรียกเคอร์เนล์ฟังก์ชันบนเวิร์กเกอร์ก็จะต้องเรียกรันไทม์ฟังก์ชัน _StreamSeqKernelRegister เพื่อเก็บข้อมูลเคอร์เนลและกำหนดค่าเริ่มต้น

ฟังก์ชัน _StreamSeqEnqueueScatterND และ _StreamSeqEnqueueGatherND ใช้สำหรับกำหนดตัวแปรที่ต้องการแบ่งการกระจายและรวบรวมข้อมูลบนเคอร์เนลนี้ ทั้งคู่จะต้องเรียกก่อนการประมวลผลเคอร์เนล ฟังก์ชัน _StreamSeqKernelGetNextSequence รับผิดชอบในด้านการส่งข้อมูลแบบไม่กีดกันไปยังดีไวซ์ เพื่อที่จะทำงานไปพร้อมกันกับการประมวลผล มีขั้นตอนการทำงานอย่างคร่าวคือ

1. กระจายข้อมูลอินพุตแบบสแคทเทอร์สำหรับเคอร์เนลย่อยส่วนที่ i+2 แบบไม่กีดกันการส่ง ด้วยฟังก์ชัน MPI_Iscatter
2. ส่งข้อมูลอินพุตสำหรับเคอร์เนลย่อยส่วนที่ i+1 แบบไม่กีดกันการส่งไปยังดีไวซ์ด้วยฟังก์ชัน clEnqueueWriteBuffer ที่กำหนดค่า blocking เป็น CL_FALSE

จากนั้นโปรแกรมจะเริ่มทำการประมวลผลเคอร์เนลย่อยส่วนที่ i และเมื่อเสร็จสิ้นจะเรียกฟังก์ชัน _StreamSeqKernelFinishSequence เพื่อเริ่มการส่งกลับข้อมูลเอาต์พุตแบบไม่กีดกันการส่ง ฟังก์ชัน _StreamSeqKernelFinishSequence มีขั้นตอนการทำงานคือ

1. ส่งข้อมูลเอาต์พุตสำหรับเคอร์เนลย่อยส่วนที่ i-1 จากดีไวซ์กลับมายังโฮสแบบไม่กีดกัน ด้วยฟังก์ชัน clEnqueueReadBuffer ที่กำหนดค่า blocking เป็น CL_FALSE

2. รวมข้อมูลเอาต์พุตแบบแกทเธอร์สำหรับเคอร์เนลย่อยส่วนที่ i-2 แบบไม่กีดกันการส่ง ด้วยฟังก์ชัน MPI_Igather

รันไทม์ไลบรารี่ฟังก์ชันเพิ่มเติมสำหรับการปรับปรุงสมรรถนะทั้งหมดอยู่ในตารางที่ 5

ตารางที่ 5 รันไทม์ไลบรารี่ฟังก์ชันเพิ่มเติมสำหรับการปรับปรุงสมรรถนะ

ฟังก์ชัน	รายละเอียด
_StreamSeqKernelRegister	เก็บข้อมูลและกำหนดค่าเริ่มต้นให้กับเคอร์เนลก่อนเริ่มการทำงานแบบแบ่งเคอร์เนล
_StreamSeqEnqueueScatterND	กำหนดตัวแปรอินพุตที่มีการกระจายแบบสแคทเทอร์ให้กับเคอร์เนล
_StreamSeqEnqueueGatherND	กำหนดตัวแปรเอาต์พุตที่มีการรวมแบบแกทเธอร์ให้กับเคอร์เนล
_StreamSeqKernelGetNextSequence	กระจายข้อมูลอินพุตแบบสแคทเทอร์แบบไม่กีดกันล่วงหน้า ก่อนการประมวลผลเคอร์เนลย่อย
_StreamSeqExec	ตรวจสอบว่ายังต้องมีการประมวลผลเคอร์เนลย่อยอยู่อีกหรือไม่
_StreamSeqKernelFinishSequence	รวมข้อมูลเอาต์พุตแบบแกทเธอร์แบบไม่กีดกันย้อนหลัง หลังการประมวลผลเคอร์เนลย่อย
_StreamSeqFinishGatherND	บล็อครอการทำงานของการรวมข้อมูลเอาต์พุตแบบแกทเธอร์แบบไม่กีดกันทั้งหมด

สำหรับตัวอย่างโปรแกรมคูณเมทริกซ์เมื่อแปลงโค้ดแบบให้ที่มีการปรับปรุงสมรรถนะอัตโนมัติจะได้โค้ดผลลัพธ์ดังภาพที่ 22

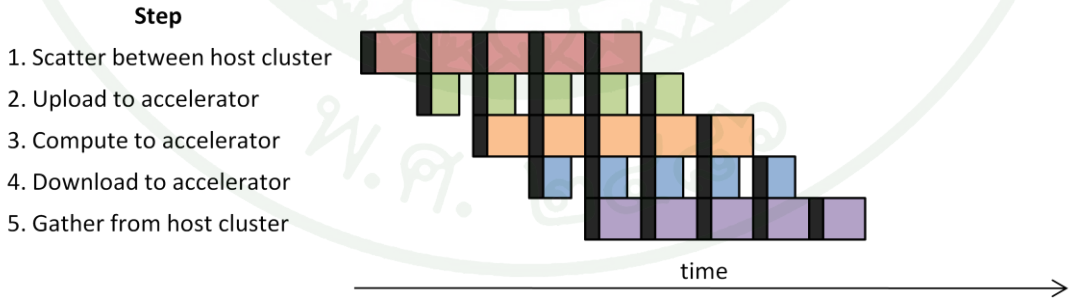
```
void _Function_670420(){
…
_StreamSeqKernelRegister(_kernel_id, _local_tid_start, _local_tid_end, 0, (nsquare) - 1, _loop_step, &_kernel_info);
/* Distribute Array Memory */
_EnqueueBroadcastND((void *) B[0], _cl_mem_B, _id_B, __TYPE_FLOAT(), 1, 1, 2, n, n);
_StreamSeqEnqueueScatterND(_kernel_info, (void *) A[0], cl_mem_A,_id_A,FLOAT_T, MEM_READ_ONLY, 0, (nsquare) - 1, 1, 0, 0, 1, 1, 2, 0, n, n);
_StreamSeqEnqueueGatherND(_kernel_info, (void *) C[0], cl_mem_C,_id_C,FLOAT_T, MEM_WRITE_ONLY), 0, (nsquare) - 1, 1, 0, 0, 1, 1, 2, 0, n, n);
_FinishDistributeArray();
while (1)
{
 int _stream_loop_size, _stream_completed = 0, _sequence_id = 0;
 size_t _stm_global_item_num, _stm_work_group_item_num;
 _StreamSeqKernelGetNextSequence(kernel_info,&_sequence_id,&_stm_global_item_num,
   &_stm_work_group_item_num);
 if (StreamSeqExec(_kernel_info->stream_seq_start_idx, _kernel_info->stream_seq_end_idx))
 {
   /* Compute Workload */
    _kernel = _CreateKernel("_kernel_670420");
    _SetKernelArg(_kernel,0,sizeof(int),(void *)&nsquare);
    _SetKernelArg(_kernel,1,sizeof(int),(void *)&n);
    _SetKernelArg(_kernel,2,sizeof(int),(void *)&i);
    _SetKernelArg(_kernel,3,sizeof(int),(void *)&j);
    _SetKernelArg(_kernel,4,sizeof(cl_mem),(void *)&_cl_mem_B);
    _SetKernelArg(_kernel,5,sizeof(cl_mem),(void *)&_cl_mem_A);
    _SetKernelArg(_kernel,6,sizeof(cl_mem),(void *)&_cl_mem_C);
    _SetKernelArg(_kernel,7,sizeof(int),(void *)&_kernel_info->stream_seq_start_idx);
    _SetKernelArg(_kernel,8,sizeof(int),(void *)&_kernel_info->stream_seq_end_idx);
    _SetKernelArg(_kernel,9,sizeof(int),(void *)&_loop_step);

_LaunchKernel(_kernel,&_stm_global_item_num,&_stm_work_group_item_num,_kernel_info);
    _ClearKernel(_kernel);
 }
 _StreamSeqKernelFinishSequence(_kernel_info);
 if (_kernel_info->is_complete) break;
}
/* Gather Array Memory */
 _StreamSeqFinishGatherND(_kernel_info, (void *)C[0],_cl_mem_C,_id_C, FLOAT_T, MEM_WRITE_ONLY, 0, (nsquare) - 1, 1, 0, 0, 1, 1, 2, 0, n, n);
_FinishGatherArray();
```

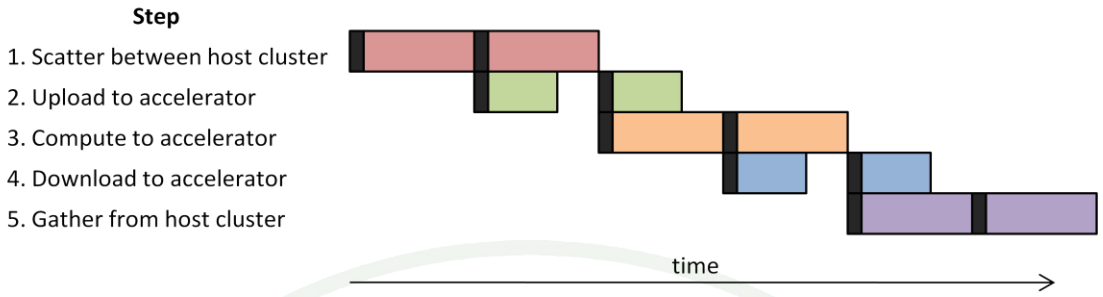
ภาพที่ 22 ตัวอย่างผลลัพท์ของการปรับสมรรถนะ

7.3 การหาเวลาที่ดีที่สุด การปรับปรุงสมรรถนะ ด้วยวิธีนี้สามารถแบ่งเคอร์เนลได้เป็นหลายขนาด เช่น ถ้ามีลูปที่มีรอบการทำงาน 1,000 รอบ อาจจะแบ่งไปทำงานครั้งละ 50 รอบ ซึ่งก็จะต้องทำซ้ำ 20 ครั้งจึงจะเสร็จสมบูรณ์ หรือถ้าแบ่งเป็นครั้งละ 100 รอบ ก็จะต้องทำงาน 10 ครั้ง การแบ่งจำนวนรอบที่ไม่เท่ากันจะได้เวลาในการประมวลผลรวมไม่เท่ากันไปด้วย เนื่องจากแต่ละครั้งของการแบ่งการทำงานจะเกิดโอเวอร์เฮดต่างๆ เช่น สำหรับการกระจายข้อมูลไปยังแต่ละโหนดก็จะต้องมีการเตรียมบัฟเฟอร์ในการส่ง การส่งข้อมูลลงไปยังเน็ตเวิร์กเป็นต้น ดังนั้นหากแบ่งด้วยขนาดที่เหมาะสมก็จะส่งผลให้เวลาการประมวลผลรวมรวดเร็ว

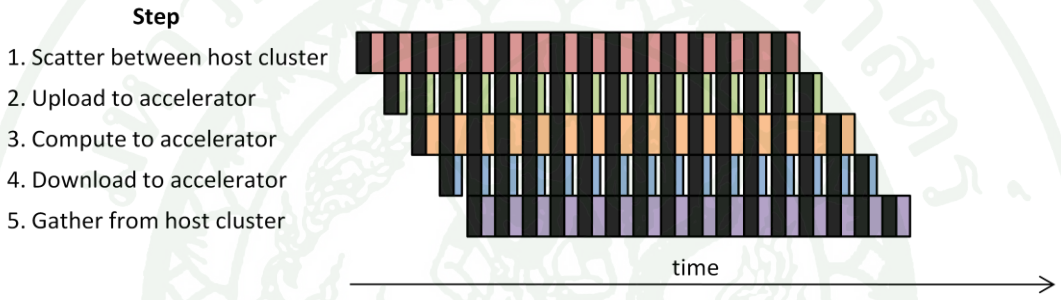
ภาพที่ 23 เป็นการแบ่งเคอร์เนลออกเป็น 5 ส่วน หรือถ้าสมมติให้เคอร์เนลนี้มีรอบการทำงานของลูปเป็น 800 รอบ ก็จะได้ว่าขนาดของเคอร์เนลย่อยคือ 160 ส่วนสีดำในแต่ละขั้นตอนคือโอเวอร์เฮดที่เกิดขึ้นของแต่ละรอบการทำงาน เมื่อเปรียบเทียบกับ ภาพที่ 24 ที่ซึ่งแบ่งเคอร์เนลออกเป็น 2 ส่วน (และถ้าสมมติให้มีรอบการทำงาน 800 รอบ ก็จะหมายถึงขนาดของการแบ่งเป็น 400) จะพบว่าเวลาการทำงานรวมของภาพที่ 23 นั้นเร็วกว่า เนื่องจากมีส่วนการทำงานที่ทำงานขนานกันมากกว่า แต่นั่นไม่ได้หมายถึงว่าขนาดของเคอร์เนลย่อยยิ่งเล็กยิ่งดี เมื่อพิจารณา ภาพที่ 25 ซึ่งแบ่งเคอร์เนลด้วยขนาดที่เล็กมาก จะพบว่าเวลาการทำงานเพิ่มขึ้นทั้งทีมีเวลาการทำงานแบบขนานมากกว่า แต่เวลารวมก็ถูกเพิ่มขึ้นเนื่องจากเวลาที่เป็นโอเวอร์เฮดถูกสะสมรวมเพิ่มขึ้น ดังนั้นผู้วิจัยจึงพยายามหาวิธีการหาขนาดของเคอร์เนลย่อยที่ดีที่สุด เพื่อให้เวลาการทำงานหลังจากการปรับสมรรถนะด้วยวิธีนี้มีประสิทธิภาพมากที่สุด



ภาพที่ 23 การแบ่งเคอร์เนลด้วยขนาดที่เหมาะสม



ภาพที่ 24 การแบ่งเคอร์เนลด้วยขนาดที่ใหญ่เกินไป



ภาพที่ 25 การแบ่งเคอร์เนลด้วยขนาดที่เล็กเกินไป

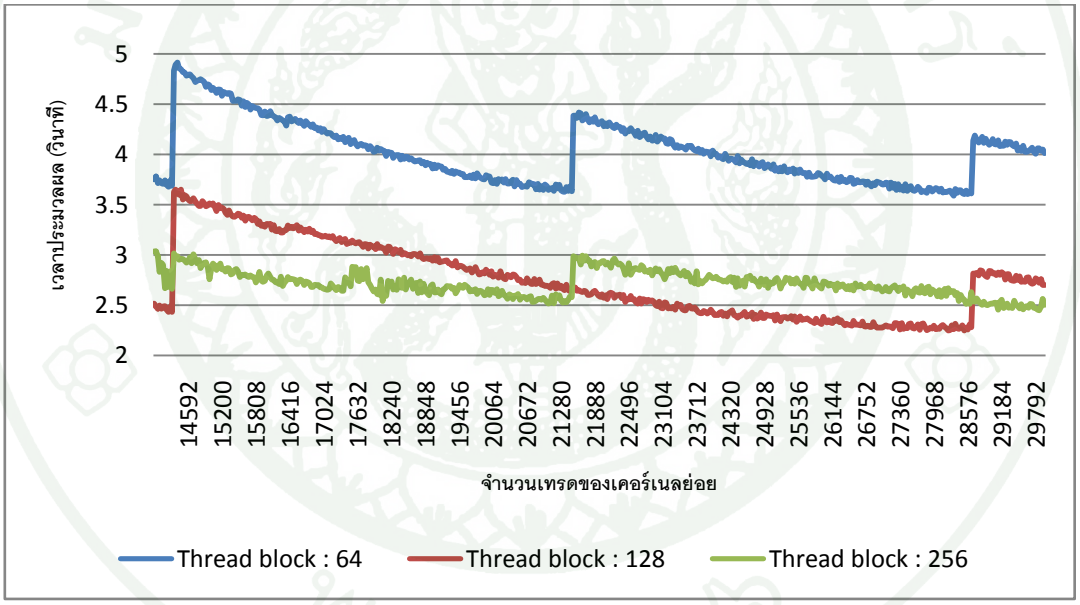
เวลาการประมวลผลรวมของโปรแกรมที่ทำงานบนจีพียูคลัสเตอร์ประกอบขึ้นจากเวลาการทำงานของทั้งห้าส่วนต่อไปนี้

1. เวลาการกระจายข้อมูลแบบสแคทเทอร์ไปยังแต่ละโหนด
2. เวลาการส่งข้อมูลจากโฮสไปยังดีไวซ์
3. เวลาการประมวลผลบนจีพียู
4. เวลาการส่งข้อมูลจากดีไวซ์ไปยังโฮส
5. เวลาการรวบรวมข้อมูลระหว่างโหนดแบบแกทเธอร์

ผู้วิจัยได้ทำการวิเคราะห์เวลาของทั้งห้าส่วนเพื่อหาขนาดของการแบ่งเคอร์เนลที่ดีที่สุด โดยเริ่มจากการวิเคราะห์เวลาการประมวลผลบนจีพียูก่อน ปัจจัยที่ส่งผลต่อเวลาการประมวลผลบนจีพียู ได้แก่ จำนวนเทรดทั้งหมด ขนาดของเทรดบล็อค และรูปแบบในการเข้าถึงข้อมูล (Memory Access Pattern) เป็นต้น ขนาดของเทรดบล็อคที่ใหญ่จะช่วยให้มีเทรดที่เป็นอิสระ

ต่อกันจำนวนมาก ช่วยเพิ่มการทำงานแบบขนาน แต่เนื่องจากหน่วยความจำรีจิสเตอร์ (Register) ต่อหนึ่ง SM มีจำนวนจำกัด การกำหนดเทรดบล็อคใหญ่เกินไปจะทำให้แต่ละเทรดต้องใช้งานหน่วยความจำโกลบอลเพิ่มขึ้น แทนจำนวนหน่วยความจำรีจิสเตอร์ที่ไม่เพียงพอ ซึ่งจะใช้เวลาเข้าถึงนานกว่า

การทดลองเบื้องต้นอันดับแรกทำด้วยการทดสอบว่า ที่ขนาดของเคอร์เนลย่อยต่างๆ เวลาการประมวลผลที่เกิดขึ้นจะเป็นอย่างไร ในการทดลองนี้ใช้โปรแกรม matmul ที่ขนาดเมทริกซ์ 4096 x 4096 และรันบนหนึ่งโหนดของจีพียูคลัสเตอร์ เพื่อไม่ให้เวลาการกระจายข้อมูลและรวบรวมข้อมูลมีผลต่อเวลารวม ทดสอบด้วยเทรดบล็อคขนาด 64, 128 และ 256 และทดลองปรับขนาดของเคอร์เนลย่อย โดยเริ่มต้นที่ 14,016 และเพิ่มค่าทีละ 32 จนถึง 29,984



ภาพที่ 26 เวลาประมวลผลเมื่อปรับจำนวนเทรดในเคอร์เนลย่อย

สำหรับสาเหตุของการที่ต้องทำการเพิ่มขนาดเคอร์เนลย่อยทีละ 32 และทดสอบกับขนาดของเคอร์เนลย่อยที่เป็นจำนวนเท่าของ 32 นั่นก็เพราะจีพียูมี วาร์ปไซส์ (Warp Size) ขนาดเท่ากับ 32 ซึ่งเป็นขนาดของเทรดที่ทำงานพร้อมกันได้สูงสุดต่อ 1 SM ผลลัพธ์ของการทดลองที่ได้เป็นไปดังภาพที่ 26 นั่นคือมีบางขนาดของเคอร์เนลย่อยที่ทำให้เวลาการประมวลผลบนจีพียูรวดเร็วมาก และบางขนาดก็ช้ากว่ากันถึงเท่าตัว จากการทดลองสำหรับที่เทรดบล็อคขนาดต่างๆ จำนวนของเทรดในเคอร์เนลย่อยที่ทำให้การประมวลผลทำได้เร็วที่สุด มีค่าตามตารางที่ 6

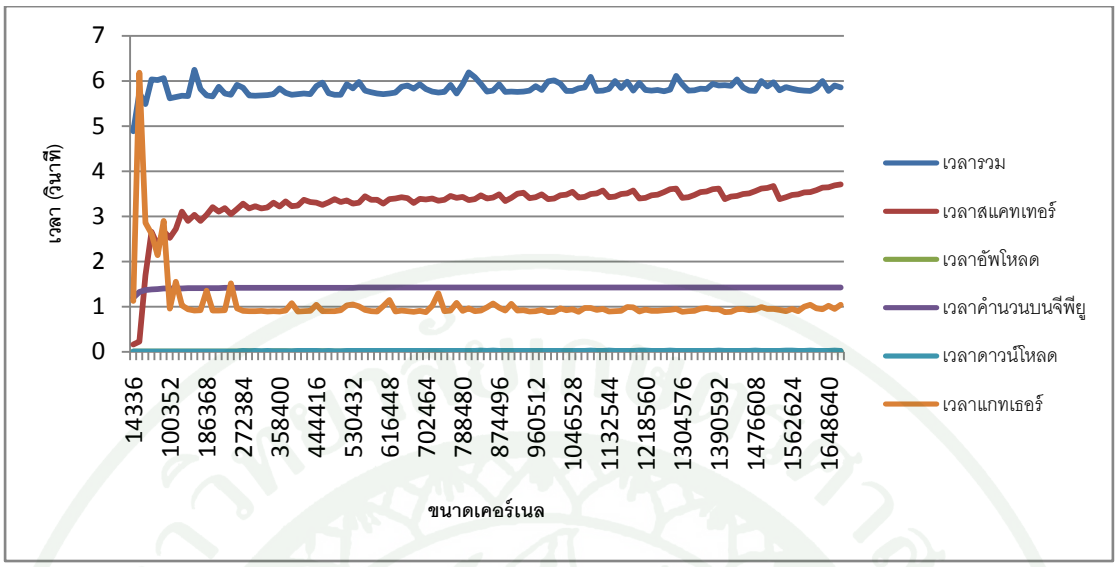
ตารางที่ 6 จำนวนเทรดในเคอร์เนลย่อยที่เหมาะสมที่สุดของแต่ละขนาดเทรดบล็อค

ขนาดเทรดบล็อค	จำนวนเทรดในเคอร์เนลย่อยที่เหมาะสมที่สุด
64	14336, 21504 และ 28672
128	14336 และ 28672
256	21504

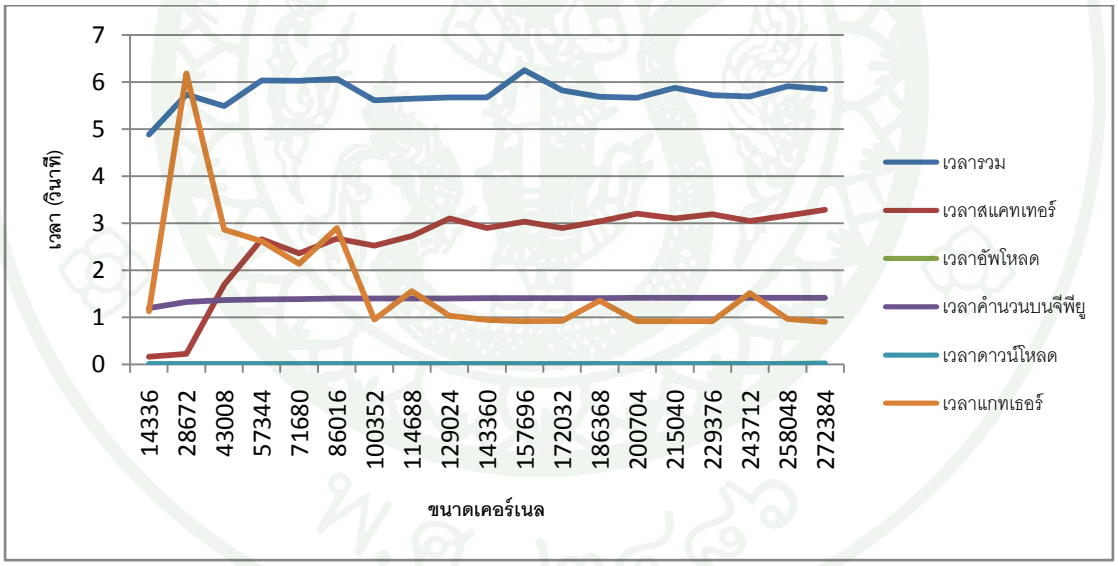
จะพบว่าจำนวนเทรดทั้งหมดต่อการทำงานหนึ่งครั้งปกติจะไม่มีผลต่อเวลาการทำงานรวมมากนัก แต่เมื่อใช้เทคนิคการแบ่งเคอร์เนล เวลาการทำงานต่างกันโดยสิ้นเชิงเมื่อขนาดเคอร์เนลย่อยไม่เท่ากัน ในการทดลองครั้งนี้ใช้กราฟิกการ์ด NVIDIA Tesla M2050 ซึ่งมีจำนวน SM มีเท่ากับ 14 หน่วย และแต่ละ SM มีจำนวน 8 คอร์ จำนวนเทรดหรือเคอร์เนลย่อยที่เหมาะสมที่สุดตามทฤษฎีคือเป็นจำนวนเท่าของผลคูณระหว่างขนาดของเทรดบล็อคและขนาดสูงสุดที่เป็นไปได้ของกริด ซึ่งขนาดสูงสุดที่เป็นไปได้ของกริดคำนวนได้จากจำนวน SM คูณด้วย จำนวนคอร์ต่อ SM นั่นคือจะต้องแบ่งขนาดของเคอร์เนลย่อยให้เท่ากับจำนวนเท่าของสมการที่ (1)

$$Thread\ block\ size\ \times Number\ of\ SM\ \times Cores\ per\ SM \qquad (1)$$

ซึ่งตรงกับการทดลองเบื้องต้นพอดี สำหรับงานวิจัยนี้จะกำหนดให้เทรดบล็อคที่ใช้มีขนาดคงที่อยู่ที่ 128 เทรด ดังนั้นจำนวนของเทรดที่เหมาะสมในการประมวลผลจึงต้องเท่ากับจำนวนเต็ม ที่เป็นจำนวนเท่าของ 14,336 (โดยที่ไม่นับรวมศูนย์) จากนั้นทำการทดลองโดยใช้โปรแกรม matmul ที่มีขนาดของเมทริกซ์เป็น 4096 x 4096 มาจับเวลาการประมวลผลบนคลัสเตอร์ขนาด 4 โหนด โดยจับเวลารวมของทั้งโปรแกรม เวลาในส่วนของการสแคทเทอร์ เวลาการส่งข้อมูลจากโฮสไปยังดีไวซ์ เวลาการทำงานบนจีพียู เวลาการส่งข้อมูลจากดีไวซ์ไปยังโฮส และเวลาการแกทเธอร์ ทดลองเปลี่ยนขนาดของเคอร์เนลย่อยตั้งแต่ 14,336 จนถึง 1,677,312 โดยเพิ่มทีละ 14,336 ผลลัพธ์เป็นไปตามภาพที่ 27 และแสดงเฉพาะขนาด 14,336 ถึง 272,384 ตามภาพที่ 28



ภาพที่ 27 เวลาการทำงานของ matmul เมื่อแบ่งออกเป็นส่วน



ภาพที่ 28 เวลาการทำงานของ matmul เมื่อแบ่งออกเป็นส่วน (เฉพาะขนาด 14,336 ถึง 272,384)

ผลลัพธ์แสดงให้เห็นว่าเวลาของการส่งข้อมูลระหว่างโฮสกับดีไวซ์นั้นน้อยมาก (เวลาอัพโหลดและเวลาดาวน์โหลดใน ภาพที่ 27) เมื่อเทียบกับการประมวลผลบนจีพียู หรือเวลาการส่งข้อมูลระหว่างโหนด เนื่องจากปริมาณการรับ ส่งข้อมูลของจีพียูสูงมากเมื่อเทียบกับเน็ตเวิร์ก ดังนั้นในการวิเคราะห์หาขนาดเคอร์เนลย่อยที่ดีที่สุดจึงสามารถตัดปัจจัยเรื่องเวลาการส่งข้อมูลระหว่างโฮสและดีไวซ์ออกไปได้

เวลาการประมวลผลรวมสามารถประมาณได้จากค่าสูงสุดของเวลาแสคทเทอร์รวม เวลาแกทเธอร์รวม และเวลาประมวลผลบนจีพียูรวมดังสมการที่ (2)

$$Exec.Time \approx MAX(Scatter\ Time_{total}, GPU\ Exec.Time_{total}, Gather\ Time_{total}) \quad (2)$$

ผลการทดลองเบื้องต้นแสดงให้เห็นว่าการประมาณค่าด้วยวิธีนี้มีความถูกต้อง ทั้งนี้สังเกตได้จากเวลารวมจะเปลี่ยนแปลงไปตามเวลาที่มีค่ามากที่สุดของเวลาทั้งสามค่า ในส่วนของเวลาที่หายไป ซึ่งทำให้เวลาการทำงานทั้งหมดรวมกันแล้วมีค่าไม่เท่ากับเวลารวม เกิดขึ้นจาก 1) เวลาของการกระจายข้อมูลสเกลาร์ 2) เวลาของการจองและคืนพื้นที่หน่วยความจำ 3) เวลาในการกำหนดค่าในรันไทม์ไลบรารีรวมถึงการคำนวนค่าเริ่มต้น และ 4) เวลาในการ บรอดคาซท์เมทริกซ์ B เวลาทั้งหมดเป็นค่าคงตัวที่ไม่ขึ้นกับขนาดของเคอร์เนลย่อย และมีค่าน้อยมาก ยกเว้นเวลาในการบรอดคาซท์เมทริกซ์ B ซึ่งใช้เวลาถึง 2.4338386 วินาที

เวลาของการแสคทเทอร์และแกทเธอร์ขึ้นอยู่กับประสิทธิภาพของเครื่องและเน็ตเวิร์กที่ทำการประมวลผล เพื่อให้ข้อมูลส่วนนี้สามารถนำมาใช้ได้ทันที และสามารถนำกลับมาใช้ได้ตลอด จึงใช้วิธีการประมวลผลก่อนเพื่อเก็บเวลาของการแสคทเทอร์และแกทเธอร์ที่ขนาดข้อมูลต่างๆ และเก็บไว้ยังไฟล์ ทุกครั้งที่โปรแกรมถูกรันขึ้นจะอ่านค่าจากไฟล์นี้ ผู้วิจัยได้ทำการสร้างโปรแกรมสำหรับเก็บข้อมูลเวลาของการแสคทเทอร์และแกทเธอร์ ด้วยการเรียกใช้งานฟังก์ชัน MPI_Iscatterv และ MPI_Igatherv ที่ข้อมูลขนาดต่างๆ เก็บข้อมูลของเวลาการทำงานแต่ละครั้ง (หน่วยเป็นวินาที) เรียกใช้งานเป็นจำนวน 50 รอบ หาค่าเฉลี่ยของเวลา และเขียนค่าของขนาดข้อมูลพร้อมเวลาลงไฟล์ ดังตัวอย่างของไฟล์ใน ภาพที่ 29 ในการทดลองเบื้องต้นจะเก็บเวลาเฉพาะขนาด 14,336 ถึง 1,677,312 ตัวของตัวแปรชนิด double (เพิ่มขึ้นทีละ 14,336) เพื่อใช้งานกับโปรแกรม matmul โดยเฉพาะ ในอนาคตสามารถปรับปรุงให้ใช้งานกับโปรแกรมอื่นได้ด้วยการเก็บจำนวนของขนาดข้อมูลที่มากขึ้น

เวลาการประมวลผลบนจีพียูประมาณได้ยากกว่า เนื่องจากไม่คงที่สำหรับแต่ละโปรแกรมประยุกต์ และไม่สามารถบอกอัตราการโตของเวลาเมื่อเทียบกับขนาดของอินพุตได้ ดังนั้นผู้วิจัยจึงใช้วิธีเก็บข้อมูลขณะรันไทม์ ด้วยการจับเวลาที่ขนาดเคอร์เนลย่อยต่างๆกัน โดยเริ่มต้นจาก 14,336 เทรด และเพิ่มขนาดทีละสองเท่าจนถึง 458,752 เทรด และใช้การปรับเส้นโค้ง (Curve Fitting) เพื่อประมาณเวลาของขนาดที่เหลือ เมื่อได้เวลารวมของแสคทเทอร์กับแกทเธอร์

และเวลาการประมวลผลบนจีพียูแล้ว ก็จะสามารถประมาณเวลาด้วยการหาค่าสูงสุดของทั้งสามค่าได้ เนื่องจากขนาดเคอร์เนลย่อยที่เป็นไปได้มีจำนวนไม่มาก ดังนั้นการแสกนหาเวลาต่ำสุดของทุกขนาดจึงทำได้รวดเร็ว

```
SCATTER
14336   0.000136800000000000
28672   0.000376280000000000
43008   0.004325759999999999
57344   0.009062399999999998
…
0       0
GATHER
14336   0.000961060000000000
28672   0.035546780000000000
43008   0.007309719999999997
57344   0.008921879999999998
…
0       0
```

ภาพที่ 29 ตัวอย่างไฟล์เก็บข้อมูลเวลาแสคทเทอร์และแกทเธอร์

ข้อจำกัดของการปรับปรุงสมรรถนะแบบนี้คือ

1. ไม่สามารถใช้งานได้กับโปรแกรมที่ใช้ขั้นตอนวิธีแบบทำซ้ำ (เช่น โปรแกรม gaussblur และ srad ที่ใช้ในการทดสอบ) เนื่องจากรอบการทำงานที่มากจะได้ประโยชน์จากการแบ่งการทำงานน้อยลง และมีโอเวอร์เฮดในการแบ่งสูงขึ้นมาก โปรแกรมที่ใช้ขั้นตอนวิธีแบบทำซ้ำ มักมีเวลาการประมวลผลที่เร็วอยู่แล้วบนจีพียูคลัสเตอร์

2. สำหรับข้อมูลที่ใช้การกระจายแบบสแคทเทอร์สามารถแบ่งได้ทันที เนื่องจากมีคลอสกำกับไว้แล้วว่าจะต้องแบ่งอย่างไร แต่ข้อมูลที่กระจายแบบบรอดคาซท์จะต้องมีการวิเคราะห์การเข้าใช้งานข้อมูลก่อนจึงจะสามารถแบ่งได้ถูกต้อง โปรแกรม matmul ที่ใช้ในการทดสอบการปรับสมรรถนะจะมีทั้งเมทริกซ์ที่ใช้การกระจายแบบสแคทเทอร์และแบบบรอดคาซท์ ดังนั้นเมทริกซ์ที่กระจายแบบบรอดคาซท์จะไม่ถูกแบ่ง แต่จะบรอดคาซท์ทั้งหมดก่อนเริ่มการทำงาน ทำให้การปรับสมรรถนะเป็นไปแบบไม่เต็มประสิทธิภาพ

ผลและวิจารณ์

จุดประสงค์ของงานวิจัยนี้คือการสร้างเฟรมเวิร์กสำหรับช่วยลดเวลาการพัฒนาโปรแกรมที่ทำงานบนจีพียูคลัสเตอร์ แต่ยังคงประสิทธิภาพการทำงานไว้ ดังนั้นในการวัดผลจะแบ่งออกเป็นสองเป้าหมายคือ ประสิทธิภาพ (Performance) และผลิตภาพ (Productivity) โดยใช้โปรแกรมเปรียบเทียบสมรรถนะ (Benchmark) มาตรฐาน 3 โปรแกรม ในการทดสอบ ได้แก่ matmul และ gaussblur จาก KernelGen compiler performance test suite และ srad จาก Rodinia benchmark suite

โปรแกรม matmul และ gaussblur จากชุดทดสอบของ KernelGen มีเวอร์ชันที่เขียนด้วยโอเพนเอซีซี ดังนั้นจึงเพิ่มเพียงตัวชี้แนะคอมไพเลอร์ที่นำเสนอขึ้นมาใหม่ในงานวิจัยนี้ ทางด้าน srad มีเฉพาะโค้ดที่เขียนด้วยโอเพนเอ็มพี แต่ก็สามารถปรับเปลี่ยนให้เป็นโอเพนเอซีซีได้โดยปรับปรุงเพียงเล็กน้อย ด้วยการเปลี่ยนแพร็กมา parallel for เป็น parallel loop เพิ่มแพร็กมา data สำหรับการส่งข้อมูล และเพิ่มคลอสที่เกี่ยวกับข้องกับข้อมูลที่ใช้ลงไป

การวัดประสิทธิภาพ

เพื่อแสดงให้เห็นว่าเฟรมเวิร์กและเทคนิคการคอมไพล์ที่นำเสนอทำงานได้อย่างถูกต้อง และทำให้โปรแกรมที่พัฒนาขึ้นทำงานได้อย่างมีประสิทธิภาพ จึงทำการทดสอบด้วยการจับเวลาการประมวลผลระหว่างโปรแกรมที่พัฒนาขึ้นเองด้วยเอ็มพีไอและโอเพนซีแอล กับโปรแกรมที่พัฒนาด้วยโมเดลที่นำเสนอ เปรียบเทียบเวลาการประมวลผลและค่าสปีดอัป (Speedup) เมื่อเทียบกับโปรแกรมเชิงลำดับ เพื่อดูว่าเทคนิคการคอมไพล์ที่ออกแบบไว้ลดประสิทธิภาพของโปรแกรมลงไปมากน้อยเพียงใด

การทดลองเริ่มจากการคอมไพล์โปรแกรมเชิงลำดับของโปรแกรมทดสอบทั้งสามด้วย gcc เวอร์ชัน 4.1.2 บนเครื่องโฮส ใช้แฟร็ก –O3 เพื่อให้ได้เวลาการทำงานที่ดีที่สุดบนโฮส สำหรับแต่ละโปรแกรม จะทดสอบด้วยพารามิเตอร์ดังนี้

1. matmul กำหนดให้ขนาดของเมทริกซ์เป็นเมทริกซ์จัตุรัส (sx/sy/sz) ที่มีขนาด 1024 x 1024, 2048 x 2048, 4096 x 4096 และ 8192 x 8192 ทำงานซ้ำเป็นจำนวน (iteration) 1 รอบ

2. gaussblur กำหนดให้ขนาดของปัญหาเป็น (nx/ny) 1024 x 1024, 2048 x 2048, 4096 x 4096 และ 8192 x 8192 ทำงานซ้ำเป็นจำนวน (nt) 512 รอบ

3. srad กำหนดให้ขนาดของปัญหาเป็น 1024 x 1024, 2048 x 2048 และ 4096 x 4096 ทำงานซ้ำเป็นจำนวน 64 รอบ และค่าแลมบ์ดา เป็น 0.5 โปรแกรมทดสอบนี้ไม่สามารถทดสอบด้วยขนาดของปัญหา 8192 x 8192 เนื่องจากใช้หน่วยความจำเกินกว่าขนาดหน่วยความจำของจีพียู

เขียนโปรแกรมทดสอบทั้งสามให้สามารถรันบนจีพียูคลัสเตอร์ โดยใช้เอ็มพีไอและโอเพนซีแอล โปรแกรมทดสอบที่เขียนขึ้นเองนี้เพื่อให้สำหรับเป็นโปรแกรมมาตรฐานในการเปรียบเทียบกับโปรแกรมที่สร้างขึ้นจากเฟรมเวิร์ก ใช้ mpicc ในการคอมไพล์ ซึ่งจะเรียก gcc เวอร์ชัน 4.1.2 ต่อและใช้แฟร็ก –O3 เช่นเดียวกับโปรแกรมเชิงลำดับ เนื่องจากโค้ดที่เขียนขึ้นเองนั้นสามารถปรับสมรรถนะต่อไปได้เรื่อยๆไม่มีจำกัด แต่เป้าหมายการงานวิจัยนี้ไม่ได้อยู่ที่การปรับสมรรถนะของโปรแกรมที่เขียนขึ้นเอง ดังนั้นจึงได้กำหนดข้อจำกัดในการปรับสมรรถนะของโปรแกรมที่เขียนขึ้นเองดังนี้

1. ใช้ขั้นตอนวิธีเดียวกับโปรแกรมเชิงลำดับแต่เปลี่ยนให้เป็นขั้นตอนวิธีเชิงขนาน
2. พยายามใช้หน่วยความจำในลำดับชั้นที่เร็วที่สุดของจีพียู
3. พยายามเขียนโค้ดให้การทำงานบนจีพียูกับการส่งข้อมูลทำงานพร้อมกันได้มากที่สุดโดยไม่ต้องปรับปรุงขั้นตอนวิธี

นำโปรแกรมเชิงลำดับมาเพิ่มตัวชี้แนะคอมไพเลอร์โอเพนเอซีซี รวมไปถึงส่วนต่อขยายที่นำเสนอ จากนั้นคอมไพเลอร์ด้วยซอร์สทูซอร์สคอมไพเลอร์ที่สร้างขึ้น ซึ่งจะแปลงโปรแกรมเป็นภาษาซีที่มีการเรียกรันไทม์ไลบรารีสำหรับการทำงานบนจีพียูคลัสเตอร์ โค้ดภาษาซีที่ได้จะถูกคอมไพล์ด้วย mpicc และใช้แฟร็ก –O3 จากนั้นจับเวลาการทำงานของโปรแกรมทดสอบทั้งสามโปรแกรม และสามชนิดการทำงาน สำหรับเวอร์ชันที่ทำงานบนจีพียูคลัสเตอร์จะทดลองกับการทำงานบนเครื่อง 1 โหนด 2 โหนด และ 4 โหนดเพื่อวัดความสามารถในการขยายตัวของการประมวลผล

การตรวจสอบความถูกต้องของโปรแกรมที่ทำงานดูได้จากค่าเฉลี่ยของเมทริกซ์ผลลัพธ์ของโปรแกรมทดสอบว่าเท่ากับค่าเฉลี่ยสุดท้ายของโปรแกรมเชิงลำดับหรือไม่ โปรแกรมทดสอบทั้งสามมีโค้ดในส่วนของการคำนวนค่าเฉลี่ยไว้แล้ว จึงไม่ต้องพัฒนาเพิ่มเติม ทุกโปรแกรมจะถูก

ทดสอบโดยการจับเวลาการทำงานทั้งหมดห้าครั้ง และนำเวลาทั้งหมดที่ได้มาหาค่าเฉลี่ยเลขคณิต เวลาการทำงานทั้งหมดแสดงในตารางที่ 7

ตารางที่ 7 เวลาการทำงานของโปรแกรมทดสอบ

โปรแกรมทดสอบและขนาดปัญหา		เวลาของโปรแกรมเชิงลำดับ	เวลาของโปรแกรมบนจีพียูคลัสเตอร์ที่เขียนขึ้นเอง			เวลาของโปรแกรมบนจีพียูคลัสเตอร์ที่ใช้โอเพนเอซีซีและเฟรมเวิร์กที่นำเสนอ		
			1 โหนด	2 โหนด	4 โหนด	1 โหนด	2 โหนด	4 โหนด
matmul	1024	10.1687	0.0844	0.1845	0.2700	0.1070	0.2020	0.2925
	2048	96.2181	0.7260	0.9496	1.1918	0.7956	1.0240	1.2756
	4096	1027.9066	5.8691	5.2531	5.5080	6.1452	5.5951	5.8762
	8192	11808.3314	46.6872	32.6752	27.9093	47.8556	33.9792	29.2976
gaussblur	1024	12.5918	0.6016	0.7138	0.8253	0.6652	0.8902	0.9704
	2048	50.8667	2.2084	1.7458	1.5524	2.3888	2.2467	2.0985
	4096	203.7348	8.4621	5.4324	4.0692	9.0822	7.2341	6.3701
	8192	859.2104	34.4950	20.6322	14.2352	36.9811	27.5414	23.4449
srad	1024	3.8422	0.1967	0.2782	0.3042	0.2625	0.3538	0.3810
	2048	14.9871	0.6312	0.7164	0.7566	0.8408	0.8838	0.9207
	4096	59.5263	2.3304	2.4875	2.5418	3.0968	3.1187	3.0960

สปีดอัปของโปรแกรมเชิงขนานหาได้จากการนำเวลาการทำงานของโปรแกรมเชิงขนานหารด้วยเวลาการทำงานของโปรแกรมเชิงลำดับ ตัวเลขยิ่งมากหมายถึงความเร็วที่เพิ่มขึ้นจากโปรแกรมเชิงลำดับ สปีดอัปของโปรแกรมจากโค้ดที่สร้างขึ้นด้วยคอมไพเลอร์และสปีดอัปของโค้ดที่เขียนด้วยตนเอง บนเครื่องคลัสเตอร์ขนาด 1 , 2 และ 4 โหนด ของโปรแกรมทดสอบ matmul, gaussblur และ srad เป็นไปตามตารางที่ 8

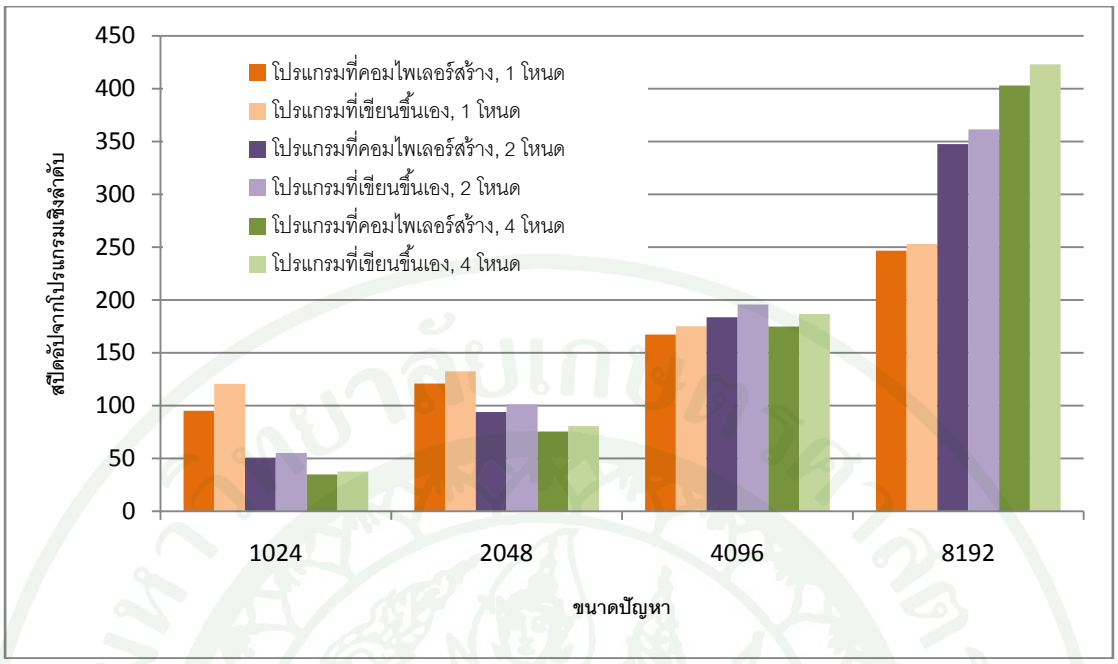
ตารางที่ 9 แสดงเปอร์เซ็นต์ของ สปีดอัปของโค้ดที่สร้างจากคอมไพเลอร์เทียบกับ สปีดอัปของโปรแกรมที่เขียนขึ้นเอง เพื่อเป็นการแสดงให้เห็นว่าเทคนิคการแปลงโค้ดที่ใช้ ลดประสิทธิภาพของโปรแกรมไปมากหรือไม่ เปอร์เซ็นต์ที่ต่ำหมายถึงเทคนิคการแปลงที่นำเสนอสร้างโปรแกรมที่ประสิทธิภาพต่ำเมื่อเทียบกับโปรแกรมเชิงขนานบนจีพียูคลัสเตอร์ที่เขียนขึ้นเอง ส่วนเปอร์เซ็นต์ที่สูงหมายถึงเทคนิคการแปลงไม่ทำให้เวลาการทำงานช้าลงมากเมื่อเทียบกับการเขียนด้วยตนเอง

ตารางที่ 8 สปีดอัปของโปรแกรมทดสอบ

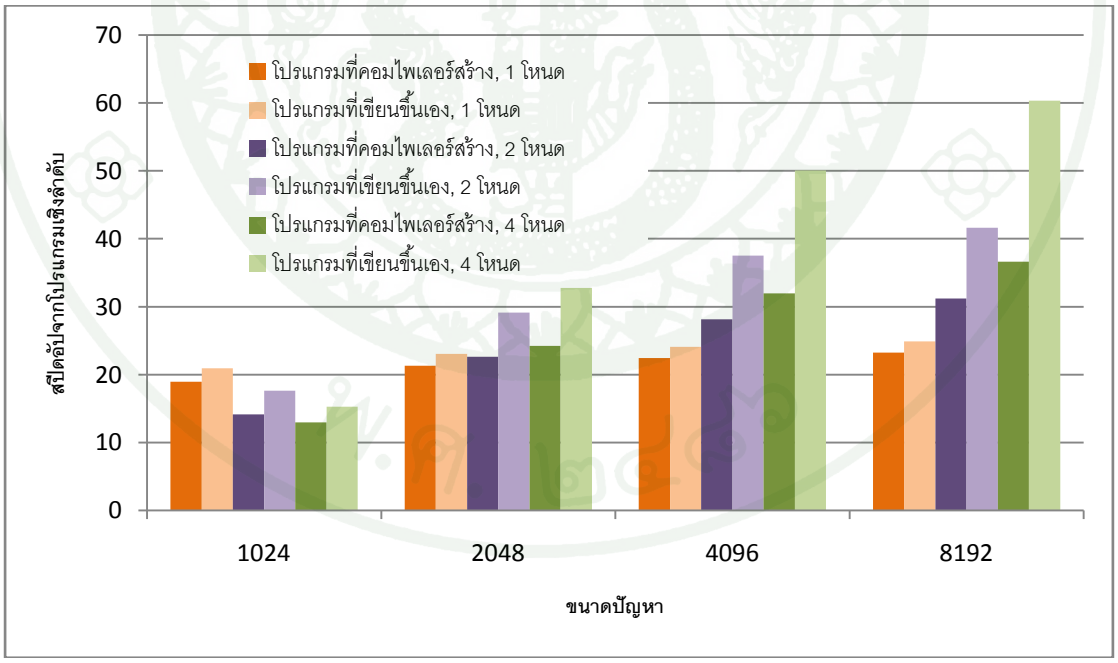
โปรแกรมทดสอบ	และขนาดปัญหา	สปีดอัปของโปรแกรมบนจีพียูคลัสเตอร์ที่เขียนขึ้นเอง			สปีดอัปของโปรแกรมบนจีพียูคลัสเตอร์ที่ใช้โอเพนเอซีซีและเฟรมเวิร์กที่นำเสนอ		
		1 โหนด	2 โหนด	4 โหนด	1 โหนด	2 โหนด	4 โหนด
matmul	1024	120.4856	55.1090	37.6669	95.0503	50.3345	34.7671
	2048	132.5402	101.3263	80.7351	120.9330	93.9620	75.4326
	4096	175.1400	195.6772	186.6206	167.2701	183.7152	174.9266
	8192	252.9246	361.3848	423.0960	246.7493	347.5170	403.0478
gaussblur	1024	20.9309	17.6410	15.2567	18.9284	14.1457	12.9763
	2048	23.0330	29.1365	32.7662	21.2934	22.6407	24.2394
	4096	24.0762	37.5038	50.0678	22.4324	28.1631	31.9829
	8192	24.9083	41.6442	60.3583	23.2338	31.1970	36.6481
srad	1024	19.5369	13.8109	12.6321	14.6344	10.8603	10.0855
	2048	23.7451	20.9205	19.8084	17.8238	16.9571	16.2786
	4096	25.5429	23.9297	23.4190	19.2217	19.0869	19.2269

ตารางที่ 9 สปีดอัปเปรียบเทียบของโปรแกรมทดสอบ

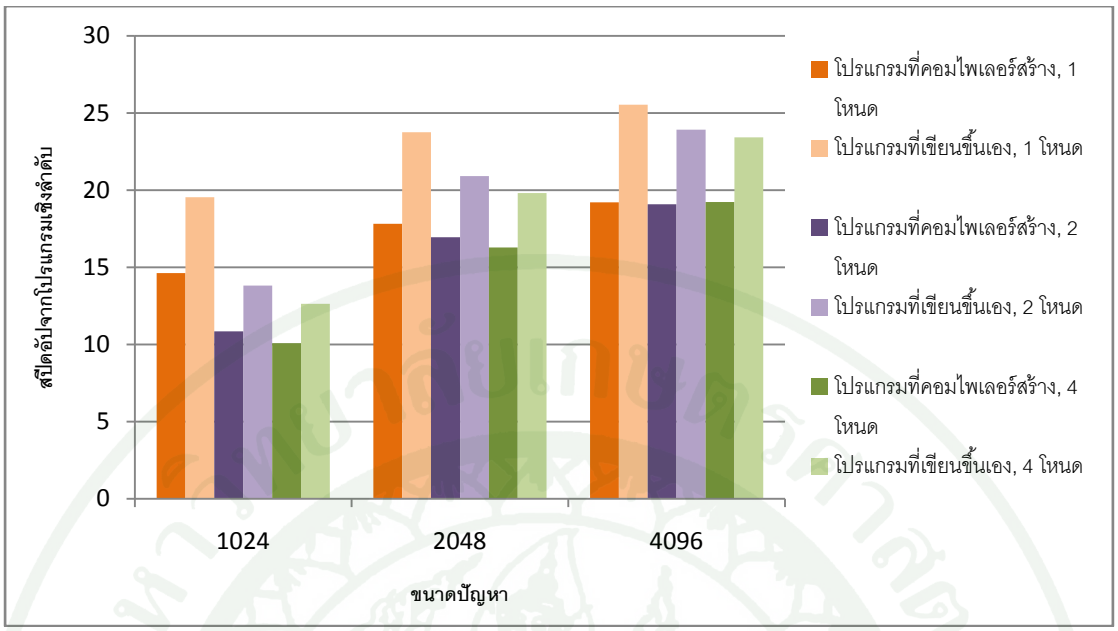
โปรแกรมทดสอบ	และขนาดปัญหา	สปีดอัปเปรียบเทียบของโปรแกรมบนจีพียูคลัสเตอร์ที่ใช้โอเพนเอซีซีและเฟรมเวิร์กที่นำเสนอกับโปรแกรมบนจีพียูคลัสเตอร์ที่เขียนขึ้นเอง		
		1 โหนด	2 โหนด	4 โหนด
matmul	1024	78.89 %	91.34 %	92.30 %
	2048	91.24 %	92.73 %	93.43 %
	4096	95.51 %	93.89 %	93.73 %
	8192	97.56 %	96.16 %	95.26 %
gaussblur	1024	90.43 %	80.19 %	85.05 %
	2048	92.45 %	77.71 %	73.98 %
	4096	93.17 %	75.09 %	63.88 %
	8192	93.28 %	74.91 %	60.72 %
srad	1024	74.91 %	78.64 %	79.84 %
	2048	75.06 %	81.06 %	82.18 %
	4096	75.25 %	79.76 %	82.10 %



ภาพที่ 30 กราฟแสดงสปีดอัปของโปรแกรมทดสอบ matmul



ภาพที่ 31 กราฟแสดงสปีดอัปของโปรแกรมทดสอบ gaussblur



ภาพที่ 32 กราฟแสดงสปีดอัปจากโปรแกรมทดสอบ srad

ภาพที่ 30 ภาพที่ 31 และภาพที่ 32 เป็นการนำข้อมูลสปีดอัปในตารางที่ 8 มาพล็อตให้อยู่ในรูปของกราฟแท่ง เพื่อให้สามารถวิเคราะห์และทำความเข้าใจได้มากขึ้น โดยในแกนนอนจะเป็นขนาดของปัญหา แกนตั้งแสดงถึงค่าสปีดอัป และพล็อตค่าสปีดอัปของโปรแกรมจีพียูคลัสเตอร์ที่เขียนขึ้นโดยใช้โอเพนเอซีซีและเฟรมเวิร์กที่นำเสนอ (โปรแกรมที่คอมไพเลอร์สร้าง) กับสปีดอัปของโปรแกรมจีพียูคลัสเตอร์ที่เขียนขึ้นเอง (โปรแกรมที่เขียนขึ้นเอง) ที่ขนาดของคลัสเตอร์โหนดเป็น 1, 2 และ 4 โหนด

โปรแกรมทดสอบ matmul เป็นโปรแกรมคูณเมทริกซ์ด้วยขั้นตอนวิธีแบบใช้แถวเป็นหลัก (Row Oriented Algorithm) ใช้เพื่อทดสอบการแบ่งลูป และการกระจายข้อมูลแบบ partition และ boardcast ของคอมไพเลอร์ จาก ภาพที่ 30 สปีดอัปของโปรแกรมที่เขียนขึ้นเองและที่คอมไพเลอร์สร้างขึ้นมีค่าสูงมากทั้งคู่ เนื่องจากค่าความซับซ้อนในการคำนวนของโปรแกรมนี้ ($O(N^3)$) มากกว่าการใช้งานหน่วยความจำ ($O(N^2)$) โปรแกรมประเภทนี้จึงเป็นโปรแกรมที่เหมาะสมกับการทำงานบนจีพียูคลัสเตอร์ ที่ขนาดของปัญหา 1024 และ 2048 การเพิ่มจำนวนโหนดของคลัสเตอร์จะยิ่งทำให้สปีดอัปของโปรแกรมลดลง เนื่องจากเวลาจากการส่งข้อมูลมากกว่าเวลาของการคำนวน เป็นผลมาจากขนาดของปัญหาที่เล็กเกินไป (การคำนวนไม่คุ้มค่าพอกับเวลาที่จะต้องเสียไปกับการกระจาย

ข้อมูล) ในขณะที่ขนาดของปัญหาที่ใหญ่มากพอ (8192) การเพิ่มจำนวนโหนดก็จะเพิ่มสปีดอัปของโปรแกรมขึ้นได้ทั้งโปรแกรมที่เขียนขึ้นเองและที่คอมไพเลอร์สร้าง

สปีดอัปของโปรแกรมที่คอมไพเลอร์สร้างไม่แตกต่างจากโปรแกรมที่เขียนขึ้นเองนัก (สำหรับทุกขนาดของปัญหา และจำนวนโหนด) เนื่องจากโปรแกรม matmul มีเคอร์เนลที่ทำงานแบบขนานเพียงส่วนเดียว โอเวอร์เฮดที่เกิดขึ้นจากการแปลงโค้ดจึงมีแค่การกระจายข้อมูลสเกลาร์และการส่งอินพุตกับเอาต์พุตจากมาสเตอร์โพรเซสไปยังรูทเวิร์กเกอร์เท่านั้น ซึ่งเป็นเวลาที่ไม่สูงมากเมื่อเทียบกับการส่งข้อมูลและการคำนวน จาก ภาพที่ 31 ขนาดข้อมูลขนาด 1024 ยังคงเป็นขนาดข้อมูลที่เล็กเกินไป การเพิ่มจำนวนโหนดจะทำให้สปีดอัปลดลง แต่ที่ขนาดข้อมูล 2048 ขึ้นไปจะเป็นขนาดข้อมูลที่เหมาะสมกับระบบคลัสเตอร์ ซึ่งเมื่อเพิ่มจำนวนโหนดสปีดอัปก็จะเพิ่มขึ้นตาม

โปรแกรมทดสอบ gaussblur เป็นโปรแกรม 25-point Gaussian blur approximation ใช้ทดสอบการทำงานของแพร็กมา data และคลอส in_pattern โปรแกรมทำการคำนวนเป็นจำนวน 512 รอบ มีการกระจายข้อมูลและรวบรวมข้อมูลเพียงครั้งเดียวในช่วงก่อนและหลังการคำนวนทั้งหมด ขนาดของปัญหาที่ทดลองสามารถใช้ประโยชน์จากจำนวนเครื่องที่เพิ่มขึ้นได้ดี ยกเว้นที่ขนาดของปัญหา 1024 ซึ่งมีขนาดเล็กเกินไป เมื่อทดลองบนเครื่องขนาดหนึ่งโหนด สปีดอัปที่ได้ไม่แตกต่างจากโปรแกรมที่เขียนขึ้นเองมาก แต่สปีดอัปจะแตกต่างจากโปรแกรมที่เขียนขึ้นเองขึ้นเรื่อยๆเมื่อเพิ่มจำนวนโหนด ปัญหานี้มาจากการ บรอดคาซท์ ข้อมูลสเกลาร์จำนวนมากในทุกรอบการคำนวน ซึ่งเป็นผลมาจากการแปลงโค้ดที่ออกแบบขึ้น (โปรแกรมที่เขียนขึ้นเองจะไม่มีเพราะทราบว่าข้อมูลเหล่านี้ไม่จำเป็นต้องส่งในทุกรอบ) ซึ่งมีแนวทางการแก้ไขในอนาคตคือ การรวมข้อมูลสเกลาร์ทั้งหมดและส่งภายในครั้งเดียว และการหาวิธีวิเคราะห์ว่าค่าใดไม่จำเป็นที่จะต้องส่งซ้ำทุกครั้ง

โปรแกรมทดสอบ srad เป็นโปรแกรมที่มีการทำงานหลายรอบเช่นเดียวกับ gaussblur แต่มีจำนวนลูปที่ทำงานแบบขนาน (เคอร์เนล) มากกว่า และมีการทำงานแบบรีดักชันเพิ่มเข้ามาในตอนต้นของแต่ละรอบ จากตารางที่ 9 และภาพที่ 32 ขนาดของปัญหาที่ทำการทดสอบทั้งหมดยังคงเล็กเกินไปที่จะใช้ประโยชน์จากเครื่องคลัสเตอร์ แต่ก็ไม่สามารถเพิ่มขนาดของปัญหาได้อีก เนื่องจากโปรแกรมทดสอบใช้การจองพื้นที่ให้กับเมทริกซ์จำนวนมาก ซึ่งถ้าหากการแปลงโค้ดที่พัฒนาทำการจองหน่วยความจำบนจีพียูเฉพาะที่มีการใช้ในแต่ละโหนดก็จะสามารถเพิ่มขนาดของปัญหาขึ้นไปได้ และจะทำให้เกิดประโยชน์จากการใช้เครื่องจีพียูคลัสเตอร์สูงสุด นี่เป็นอีกจุดด้อย

ที่ควรจะศึกษาและพัฒนาต่อ ในด้านสปีดอัปพบว่าโปรแกรมที่คอมไพเลอร์สร้างขึ้นทำงานได้เร็วใกล้เคียงกับโปรแกรมที่เขียนขึ้นเอง (อย่างน้อย 74% ของสปีดอัป) แต่ยังไม่ดีเท่าโปรแกรม matmul เนื่องจากโปรแกรมทดสอบ srad มีเคอร์เนลจำนวนมาก จึงเกิดการส่งข้อมูลสเกลาร์และโอเวอร์เฮดอื่นของรันไทม์ไลบรารี

เฟรมเวิร์กที่สร้างขึ้นตั้งอยู่บนสมมติฐานที่ว่าแต่ละเครื่องของคลัสเตอร์มีความสามารถในการประมวลผลเท่าๆกัน หรือที่เรียกว่าโมเดลแบบ โฮโมจิเนียส (Homogeneous) ดังนั้นการแบ่งงานระหว่างเครื่องจึงใช้วิธีการแบ่งให้เท่าๆกันเนื่องจากเป็นวิธีที่ง่ายและมีประสิทธิภาพ อย่างไรก็ตามวิธีการที่นำเสนอสามารถใช้ได้กับเครื่องแบบ เฮทเทอโรจิเนียส (Heterogeneous) เช่นกัน ซึ่งหากนำไปใช้โดยตรง เวลาการทำงานที่ได้จะขึ้นกับความสามารถในการประมวลผลของเครื่องที่มีประสิทธิภาพต่ำที่สุด แต่ก็สามารถปรับปรุงวิธีการให้ดีขึ้นได้ด้วยการแบ่งภาระงานให้แต่ละเครื่องแบบไม่เท่ากัน โดยอาศัยการเก็บข้อมูลสมรรถนะของแต่ละโหนดไว้ก่อน จากนั้นกระจายงานไปตามขีดความสามารถของแต่ละเครื่อง

การประยุกต์ใช้งานกับเครื่องหลายจีพียูก็สามารถทำได้เช่นกัน เนื่องจากเอ็มพีไอสามารถรันแบบหลายโพรเซสบนเครื่องเดียวได้ ดังนั้นเพียงแต่ปรับปรุงให้แต่ละโพรเซสทำงานร่วมกับจีพียูแต่ละตัวบนเครื่อง โดยอาจจะใช้การจับคู่ระหว่างหมายเลขของโพรเซสและหมายเลขของจีพียูบนแพลตฟอร์ม เช่น โพรเซส 0 ใช้งานจีพียูหมายเลข 0 โพรเซส 1 ใช้งานจีพียูหมายเลข 1 เป็นต้น แต่ประสิทธิภาพที่ได้อาจจะห่างจากประสิทธิภาพสูงสุดที่เป็นไปได้ เนื่องจากเครื่องหลายจีพียูมีคุณสมบัติของการส่งข้อมูลระหว่างหน่วยความจำของจีพียูโดยไม่ผ่านหน่วยความจำหลัก (Peer-to-peer memcopies) จึงจำเป็นที่จะต้องมีการปรับปรุงรันไทม์ไลบรารีให้รองรับการทำงานแบบนี้เพื่อดึงประสิทธิภาพสูงสุดของระบบหลายจีพียูออกมา

การวัดผลิตภาพ

การวัดผลิตภาพทำได้ยากและจำเป็นต้องควบคุมตัวแปรจำนวนมาก สามารถทำได้โดยการนำกลุ่มบุคคลที่มีทักษะในการพัฒนาโปรแกรมใกล้เคียงกันมาทดสอบ โดยการให้กลุ่มหนึ่งพัฒนาโปรแกรมที่ทำงานบนจีพียูคลัสเตอร์ด้วยวิธีการปกติหรือใช้เอ็มพีไอและโอเพนซีแอลในการพัฒนาและอีกกลุ่มหนึ่งใช้เฟรมเวิร์กที่นำเสนอ และเทียบเวลาในการพัฒนา ซึ่งจะมีปัญหาในการหานักพัฒนาโปรแกรมบนจีพียูคลัสเตอร์ที่มีทักษะใกล้เคียงกันจำนวนมากมาทดสอบ ดังนั้นงานวิจัยนี้

จะใช้จำนวนบรรทัดของโปรแกรม (Lines of Code) เป็นตัวชี้วัดอย่างคร่าวว่าโมเดลการพัฒนาแบบใดที่ให้ผลิตภาพดีกว่า โดยจำนวนบรรทัดของโปรแกรมยิ่งน้อยจะหมายถึงผลิตภาพยิ่งมาก

การทดลองทำได้ด้วยการวัดจำนวนบรรทัดของโปรแกรมทดสอบทั้งสาม โดยเขียนโค้ดในรูปแบบโปรแกรมที่เชิงลำดับ (Sequential) โปรแกรมที่เขียนขึ้นด้วยเอ็มพีไอและโอเพนซีแอลเพื่อทำงานบนจีพียูคลัสเตอร์ และโปรแกรมที่เขียนด้วยโอเพนเอซีซีกับส่วนขยายที่นำเสนอ จากนั้นใช้โปรแกรม SLOCcount ในการวัดจำนวนบรรทัดของแต่ละโปรแกรม

ตารางที่ 10 จำนวนบรรทัดของโปรแกรมทดสอบของแต่ละวิธีการอิมพลีเมนต์

วิธีการอิมพลีเมนต์	จำนวนบรรทัดของแต่ละโปรแกรม		
	matmul	gaussblur	srad
โค้ดเชิงลำดับ	101	112	214
โค้ดที่เขียนขึ้นเองสำหรับจีพียูคลัสเตอร์	235	347	603
โค้ดที่เขียนด้วยโอเพนเอซีซี	106	127	246

ตารางที่ 10 เป็นผลจากการทดลองนับบรรทัดด้วยโปรแกรม SLOCcount ตัวเลขที่ได้แสดงให้เห็นว่าการใช้โอเพนเอซีซีเพื่อกำกับโปรแกรมจะเพิ่มจำนวนบรรทัดของโปรแกรมเพียงไม่กี่บรรทัดเท่านั้น ได้แก่บรรทัดที่กำกับแพร็กมาที่บอกถึงส่วนที่ทำงานแบบขนานได้ และแพร็กมาสำหรับการอธิบายการเคลื่อนที่ของข้อมูล ในขณะที่การเขียนด้วยเอ็มพีไอและโอเพนซีแอลด้วยตัวเองจะใช้บรรทัดเพิ่มขึ้นอย่างน้อยสองเท่า อันเป็นผลมาจากความซับซ้อนของการเขียนโค้ดเพื่อเรียกใช้งานไลบรารี่เอ็มพีไอและโอเพนซีแอล รวมไปถึงการแปลงขั้นตอนวิธีให้เป็นแบบขนาน การที่จำนวนบรรทัดของโปรแกรมเพิ่มมากขึ้น ย่อมหมายถึงโอกาสที่โปรแกรมจะมีข้อผิดพลาดสูงขึ้น และจะต้องมีการแก้ไขโปรแกรมบ่อยครั้งซึ่งจะลดผลิตภาพลงไปอย่างมาก

ประสิทธิภาพของการปรับสมรรถนะ

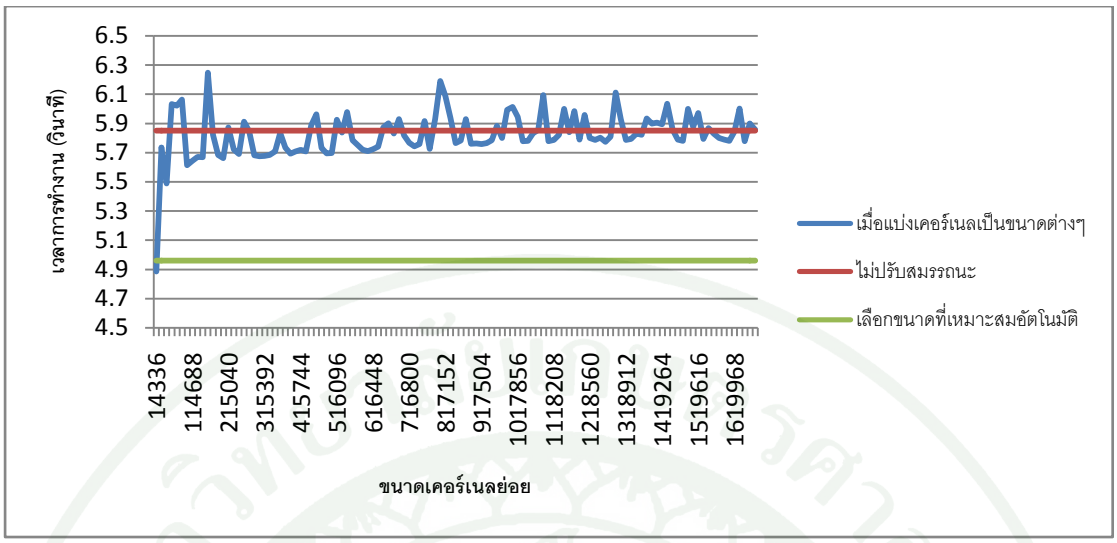
การวัดประสิทธิภาพของระบบปรับสมรรถนะอัตโนมัติ จะทำด้วยการทดลองกับโปรแกรมทดสอบสองโปรแกรมคือ matmul และ 3-moving-average โดยที่โปรแกรมทดสอบ matmul เป็นโปรแกรมคูณเมทริกซ์ที่เขียนขึ้นเอง และรวมการวนรอบของแถวและหลักของเมทริกซ์ไว้ด้วยกัน เนื่องจากข้อจำกัดของความสามารถในการปรับสมรรถนะอัตโนมัติ ใช้ชนิดตัวแปรของเมทริกซ์เป็น double ที่มีขนาด 4096 x 4096 ส่วนโปรแกรมทดสอบ 3-moving-average เป็นโปรแกรมที่จะทำการปรับปรุงค่าของสมาชิกแต่ละตัวในอาร์เรย์ ด้วยการใช้ค่าเฉลี่ยของสมาชิกที่อยู่ติดกันอีกสองตัว (ตัวก่อนหน้าและตัวถัดไป) และทดลองด้วยขนาดอาร์เรย์เท่ากับ 16,777,216

นำโปรแกรมทั้งสองมาจับเวลาการทำงานเมื่อใช้เฟรมเวิร์กที่สร้างขึ้น แต่ไม่ได้ใช้การปรับสมรรถนะ และจับเวลาการทำงานอีกครั้งเมื่อเปิดการปรับสมรรถนะอัตโนมัติ และหาสปีดอัปที่เพิ่มขึ้น ผลจากการทดลองเป็นไปดังตารางที่ 11

ตารางที่ 11 ผลการทดสอบการปรับสมรรถนะ

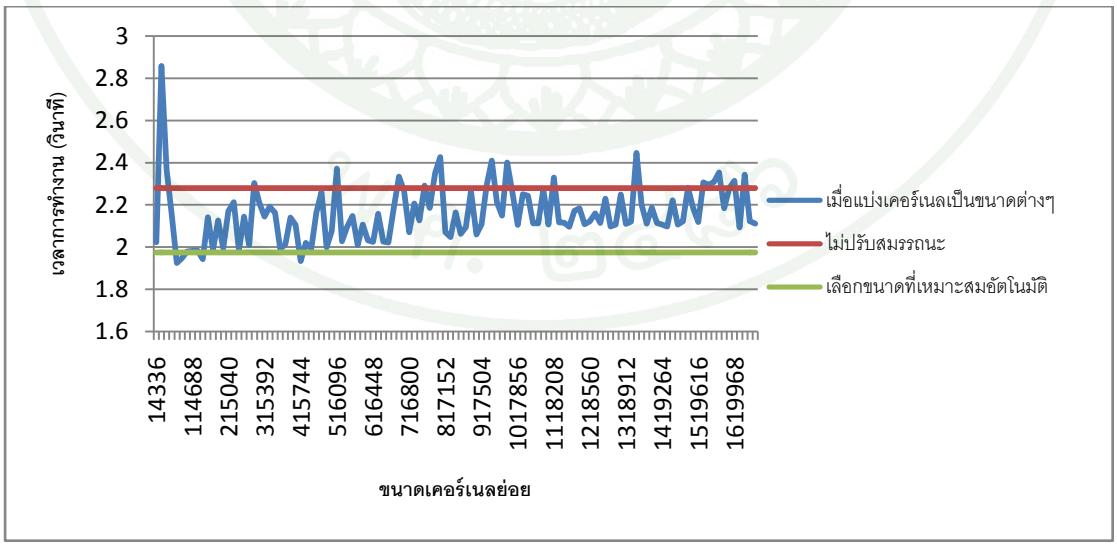
โปรแกรม	เวลาการทำงาน (ไม่ปรับสมรรถนะ)	เวลาการทำงาน (ปรับสมรรถนะ)	สปีดอัป	ขนาดของเคอร์เนลย่อยที่ดีที่สุดที่ระบบเลือก
matmul	5.849243	4.959457	1.179	14,336
3-moving-average	2.279230	1.973971	1.155	57,344

ผลการทดลองแสดงให้เห็นว่าระบบปรับสมรรถนะช่วยลดเวลาการทำงานของโปรแกรมได้ถึงแม้จะไม่มากนัก ทั้งนี้สำหรับโปรแกรม matmul มีความเป็นไปได้ที่จะลดเวลาการประมวลผลลงมาอีก หากเพิ่มการวิเคราะห์ข้อมูลและทำการแบ่งการทำงานของการ บรอดคาซท์เมทริกซ์อีกตัว ในภาพที่ 33 เป็นทดลองนำโปรแกรม matmul มาหาเวลาประมวลผลที่ขนาดเคอร์เนลย่อยต่างๆ และเปรียบเทียบกับเวลาการทำงานที่ปรับสมรรถนะแล้ว และยังไม่ปรับสมรรถนะ จากรูปจะพบว่าระบบได้เลือกขนาดเคอร์เนลย่อยที่ดีที่สุดในการทำงาน นั่นคือขนาด 14,336 แต่จะมีเวลาการทำงานช้ากว่าเล็กน้อย ซึ่งเกิดจากการทำงานของโปรแกรมที่ขนาดเคอร์เนลย่อยต่างๆกันเพื่อนำค่าไปประมาณช่วงเวลาการทำงานอื่น และเวลาเพิ่มเติมในการคำนวนสำหรับการหาขนาดที่ดีที่สุด



ภาพที่ 33 เวลาการประมวลผลของโปรแกรม matmul

เช่นเดียวกับโปรแกรม 3-moving-average ผลการทดสอบที่แสดงในภาพที่ 34 แสดงให้เห็นว่าระบบได้เลือกเวลาของขนาดเคอร์เนลย่อยที่ดีที่สุดเช่นกัน (57,344) แต่ก็มีเวลาการทำงานที่ช้ากว่าขนาดที่ดีที่สุดอยู่เล็กน้อย ถึงแม้ในโปรแกรมทดสอบระบบจะสามารถหาค่าเคอร์เนลย่อยที่ดีที่สุดได้ แต่สำหรับกรณีทั่วไปอาจจะไม่ได้ขนาดที่ดีที่สุดเสมอไป ทั้งนี้ขึ้นอยู่กับความแม่นยำในการประมาณค่าเวลาการทำงานของโปรแกรม



ภาพที่ 34 เวลาการประมวลผลของโปรแกรม 3-moving-average

การทำนายเวลาการทำงานของการติดต่อสื่อสารระหว่างกลุ่มของเอ็มพีไอ มักใช้โมเดล LogP ที่ซึ่งเป็นสมการเชิงเส้นในการประมาณ โดยเวลาของแต่ละฟังก์ชันการติดต่อสื่อสาร จะขึ้นกับค่าของเวลาแฝง (Latency) และ แบนด์วิดท์ (Bandwidth) แต่ในงานวิจัยนี้เลือกที่จะเก็บค่าของขนาดข้อมูลกับเวลาที่ทำงานโดยตรง เนื่องจากสามารถใช้ทำนายในดีกว่ากรณีที่ระบบไม่รู้ข้อมูลเกี่ยวกับฟังก์ชันการติดต่อสื่อสารระหว่างกลุ่มของเอ็มพีไอมากนัก เนื่องจากฟังก์ชันการติดต่อสื่อสารระหว่างกลุ่มของเอ็มพีไอจะมีการใช้ขั้นตอนวิธีแตกต่างกันไปตามแต่ขนาดของข้อมูลและขนาดของโหนดบนคลัสเตอร์ การเก็บข้อมูลเวลาโดยตรงทำให้สามารถทำนายเวลาได้แม่นยำกว่า และไม่ต้องทำความเข้าใจขั้นตอนวิธีที่เอ็มพีไอใช้ ซึ่งสามารถนำมาใช้ในการทำนายขนาดที่ดีที่สุดของงานวิจัยนี้ได้ เนื่องจากขนาดที่เป็นไปได้มีจำนวนจำกัด ข้อมูลที่เก็บจึงมีไม่มากนัก

สรุปและข้อเสนอแนะ

เป้าหมายหลักของงานวิจัยนี้คือการสร้างเฟรมเวิร์กที่สะดวกในการพัฒนาโปรแกรมเชิงขนานบนเครื่องคลัสเตอร์แบบระบบผสม (Heterogeneous System) ผู้วิจัยได้ใช้มาตรฐานโอเพนซีซีและสร้างส่วนขยายเพื่อให้สามารถนำมาใช้ได้กับเครื่องจีพียูคลัสเตอร์ การวิเคราะห์การกระจายข้อมูลและการติดต่อสารระหว่างเครื่องเป็นจุดที่ยากสำหรับคอมไพเลอร์ ดังนั้นผู้วิจัยได้เพิ่มส่วนขยายให้กับโอเพนเอซีซีเพื่อแก้ปัญหาการกระจายข้อมูล โดยให้ผู้พัฒนาระบุด้วยตนเองในไวยากรณ์ของสับอาร์เรย์และเพิ่มคลอส in_pattern สำหรับระบุรูปแบบการขึ้นต่อกันของข้อมูล จากนั้นทำการสร้างซอร์สทูซอร์สคอมไพเลอร์สำหรับแปลงโค้ดโอเพนเอซีซีไปยังเอ็มพีไอและโอเพนซีแอลสำหรับการทำงานบนจีพียูคลัสเตอร์

ผลการทดลองแสดงให้เห็นว่า สำหรับโปรแกรมที่มีการคำนวนจำนวนมาก (Compute Intensive Application) เมื่อมีขนาดของปัญหาที่ใหญ่พอสปีดอัปที่ได้ของโค้ดที่สร้างจากคอมไพเลอร์อย่างน้อยมีค่าเป็น 80% ของการเขียนโค้ดด้วยตนเอง ส่วนขยายของโอเพนเอซีซีที่นำเสนอนั้นก็สามารถใช้งานได้ดีกับโปรแกรมแบบ สเตนซิล (Stencil Computation) และช่วยลดเวลาในการพัฒนาไปได้มากจากการเพิ่มโค้ดเพียงไม่กี่บรรทัด

อย่างไรก็ตามยังคงต้องมีการแก้ไขเรื่องการจองพื้นที่ทั้งบนโฮสและดีไวซ์สำหรับอาร์เรย์ ทั้งนี้เทคนิคที่ใช้ช่วยลดการติดต่อสื่อสารและเพิ่มโลคอลลิตีแต่ก็ทำให้ทุกเครื่องใช้งานหน่วยความจำเหมือนกับการประมวลผลด้วยจีพียูตัวเดียว ซึ่งเป็นการจำกัดการสเกลของโปรแกรม ข้อดีของซอร์สทูซอรส์คอมไพเลอร์คือการที่ผู้พัฒนาโปรแกรมยังสามารถปรับปรุงโค้ดด้วยตัวเองได้อีกครั้ง เพื่อเพิ่มสมรรถณะของโปรแกรม นอกจากนี้ผู้วิจัยยังได้อิมพลีเมนต์รันไทม์ไลบรารี่ด้วยเอ็มพีไอและโอเพนซีแอล ทำให้สามารถย้ายไปทำงานบนแพล็ตฟอร์มตัวเร่งฮาร์ดแวร์อื่นได้

งานวิจัยนี้ใช้เทคนิคการแบ่งเคอร์เนลสำหรับการปรับปรุงสมรรถนะโปรแกรม เนื่องจากเป็นเทคนิคที่สามารถนำมาประยุกต์ใช้ได้ง่ายจากข้อมูลที่มีจากตัวชี้แนะคอมไพเลอร์และลดเวลาประมวลผลได้มาก จะได้สร้างโมเดลสำหรับการหาจุดแบ่งที่ดีที่สุด ทำให้มีประสิทธิภาพการประมวลที่เพิ่มมากขึ้น

งานวิจัยนี้ยังมีจุดที่สามารถพัฒนาต่อได้อีกมากมาย เช่น การปรับเปลี่ยนโมเดลหน่วยความจำให้ลดความซ้ำซ้อนในการจองพื้นที่ของแต่ละโหนด ซึ่งจะมีผลให้สามารถขยายการทำงานได้มากขึ้น การเพิ่มขีดความสามารถในการวิเคราะห์โปรแกรมทั้งในเวลาคอมไพล์และเวลารัน เพื่อให้สามารถใช้ได้กับโปรแกรมประยุกต์ที่หลากหลายขึ้นและมีประสิทธิภาพมากขึ้น รวมไปถึงการทดสอบและปรับปรุงเพิ่มเติมเพื่อให้ใช้งานกับ ตัวเร่งฮาร์ดแวร์ อื่นได้อย่างมีประสิทธิภาพ

เอกสารและสิ่งอ้างอิง

Basumallik, A. and R. Eigenmann. 2005. Towards automatic translation of OpenMP to MPI, pp. 189-198. *In* **Proceedings of the 19th annual international conference on Supercomputing (ICS'05)**. ACM, New York, USA.

Boyer, M., J. Men gang and K. Kumaran. 2013. Improving GPU Performance Prediction with Data Transfer Modeling, pp. 1097-1106. *In* **Proceeding of the 2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshop and PhD Forum (IPDPSW'13)**. IEEE Press.

Bueno, J., J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade and J. Labarta. 2012. Productive Programming of GPU Clusters with OmpSs, pp. 557-568. *In* **Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS'12)**. IEEE Press.

Caps Entreprise. 2013. **OpenHMPP Directives**. Available Source: http://www.caps-entreprise.com/openhmpp-directives, December 18, 2013.

Chavarria-Miranda, D., Z. Huang and Y. Chen. 2012. High-performance computing (HPC): Application & use in the power grid, pp. 1-7. *In* **Power and Energy Society General Meeting**. IEEE Press.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing, pp. 44-54. *In* **Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)**. IEEE Press.

Chen, L., L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang and B. Shou. 2010. Unified parallel C for GPU clusters: language extensions and compiler implementation, pp. 151-165. *In* **Proceedings of the 23rd international conference on Languages and compilers for parallel computing (LCPC'10)**. Springer-Verlag Press.

Ferrer, R., Royuela, S., Caballero, D., Duran, A., Martorell, X. and Ayguadé, E. 2011. **Mercurium: Design Decisions for a S2S Compiler**. Available Source: http://cetus.ecn.purdue.edu/cetusworkshop/papers/3-2.pdf, February 5, 2014.

Gaster, B.R., L. Howes, D. Kaeli, P. Mistry and D. Schaa. 2012. **Heterogeneous Computing with OpenCL**. Elsevier, Waltham, USA.

Gelado, I., J. Cabezas, N. Navarro, J. E. Stone, S. Patel and W. W. Hwu. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems, pp. 347-358. *In* **ACM SIGARCH Computer Architecture News (ASPLOS'10)**. ACM, New York, USA.

Han, T.D. and T.S. Abdelrahman. 2011. hiCUDA: High-Level GPGPU Programming, pp. 78-90. *In* **IEEE Transactions on Parallel and Distributed Systems**. IEEE Press.

KernelGen. 2013. **KernelGen Performance Test Suite**. Available Source: https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite, February 18, 2014.

Lee, S. and R. Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs, pp. 4-20. *In* **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'10)**. IEEE Press.

Lee, S., S.–J. Min, and R. Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization, pp. 101-100. *In* **Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '09)**. ACM, New York, USA.

Lee, S. and J. S. Vetter. 2012. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing, pp. 1-11. *In* **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)**. IEEE Press.

Makpaisit, P. and W. Marurnsith. 2011. Griffon – GPU Programming APIs for Scientific and General Purpose Computing, pp. 175-182. *In* **International Symposium on Distributed Computing and Artificial Intelligence (DCAI 2011)**, Springer-Verlag Press.

OpenACC. 2013. **OpenACC: Directives for Accelerator**. Available Source: http://www.openacc-standard.org, February 5, 2014.

Quinn, M.J. 2004. **Parallel Programming in C with MPI and OpenMP**. McGraw-Hill Education, Singapore.

Reyes, R., I. Lopez, J. J. Fumero and F. de Sande. 2012. accULL: An User-directed Approach to Heterogeneous Programming, pp. 654-661. *In* **Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA'12)**. IEEE Press.

Reyes, R., I. Lopez, J. J. Fumero and F. de Sande. 2012. Directive-based Programming for GPUs: A Comparative Study, pp. 410-417. *In* **Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication (HPCC'12)**. IEEE Press.

The Portland Group - PGI. 2014. **PGI Accelerator Compilers with OpenACC Directives**. Available Source: http://www.pgroup.com/resources/accel.htm, March 22, 2014.

Werkhoven B. V., J. Maasen, F.J. Seinstra and H.E. Bal. 2014. Performance models for CPU-GPU data transfer, pp. 11-20. *In* **14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)**. IEEE Press.

ประวัติการศึกษาและการทำงาน

ชื่อ นายพิสิษฐ์ มรรคไพสิฐ

เกิดวันที่ 1 กันยายน 2530

สถานที่เกิด กรุงเทพมหานคร

ประวัติการศึกษา วท.บ. (ศาสตร์คอมพิวเตอร์) มหาวิทยาลัยธรรมศาสตร์

ตำแหน่งปัจจุบัน นักพัฒนาซอฟต์แวร์

สถานที่ทำงานปัจจุบัน บริษัท พาธสเกล

ผลงานดีเด่นและ/หรือรางวัลทางวิชาการ -

ทุนการศึกษาที่ได้รับ -